Design Patterns com Java

Projeto orientado a objetos guiado por padrões

EDUARDO GUERRA

Agradecimentos Pessoais

Meu primeiro agradecimento é a Deus, por Ele ter me proporcionado tudo para que trilhasse o caminho que trilhei até chegar onde estou agora. Por ter colocado as pessoas certas para me ajudar em minhas dificuldades, por ter me colocado obstáculos nas horas adequadas que me permitiram amadurecer para superar obstáculos maiores no futuro e por ter me iluminado no momento das decisões mais críticas que acabaram direcionando minha vida no futuro.

Em seguida envio meu "muito obrigado" para meus pais, José Maria e Maria da Graça, que sempre me incentivaram e me apoiaram em minhas escolhas. Mesmo estando distantes fisicamente hoje, sei que todos os dias eles rezam e torcem por mim. Imagino direitinho eles mostrando esse livro para todo mundo com o maior orgulho! Aproveito essa oportunidade para estender esse agradecimento para toda minha familia, em especial para minha Tia Dorinha que sempre esteve do meu lado quando precisei.

Agradeço também a Maria Eduarda, a Duda, minha filha mais velha, por trazer só alegria desde que chegou nesse mundo. Seu jeito amigo e carinhoso me conforta sempre que por algum motivo estou para baixo. Não sei como tanta bondade e paciência pode caber em apenas uma pessoa. Sou muito grato pelo seu companheirismo, como quando ela senta do meu lado bem agarradinha para assistir um filme ou jogar um video-game.

Seguindo para a mais nova da familia, agradeço a Ana Beatriz, a Bia, por conter a maior concentração de energia e alegria de todo mundo. Apesar de ser bem bagunceira e gostar de um alvoroço, ela sempre faz alguma coisa que consegue tirar uma gargalhada de nossa boca nas horas mais inesperadas. É até difícil de descrever a alegria que sinto quando eu chego e essa baixinha vem gritando o meu nome e dá um abraço na minha perna.

Deixei para o fim os agradecimentos para minha companheira de toda vida, minha esposa Roberta, ou, como eu chamo ela, Lindinha. Acho que ela não tem noção

do amor que sinto por ela e como sua presença é central em minha vida. Eu agradeço a ela por através de suas ações ter se tornado a pessoa mais importante da minha vida. Sem ela eu seria incompleto e não teria estrutura para ter trilhado o caminho que segui em minha vida até esse momento. Muito obrigado por todo seu apoio, por todo seu amor e por todo seu carinho!

Agradecimentos Profissionais

Antes de mencionar alguém de forma específica, gostaria de agradecer a todos meus professores, colegas de trabalho, amigos e companheiros que de alguma forma contribuiram para meu conhecimento e me incentivaram a seguir em frente. Nossa vida é feita de pessoas e de experiências, e é a partir disso que seguimos nossos caminhos e realizamos nossas escolhas.

Primeiramente deixo um agradecimento institucional ao ITA, um instituto que é referência nacional e internacional em engenharia, em onde me formei em Engenharia da Computação e fiz meu mestrado e doutorado. Além disso, também foi um local onde tive a oportunidade de atuar como professor por mais de 5 anos. Foram experiências que me marcaram e são uma importante parte da minha história. Eu saí do ITA, mas com certeza o ITA não saiu de mim! Agradeço em especial ao professor Clovis Fernandes, meu orientador na graduação, no mestrado e no doutorado. Um grande amigo, que sempre me incentivou e que foi grande mentor, cujos ensinamentos foram muito além de questões técnicas. Espero ainda fazermos muitas coisas juntos!

Agradeço a revista MundoJ, principalmente ao Marco Guapo, inicialmente por ter aberto espaço para que escrevesse sobre minhas ideias e meus conhecimentose em seguida por ter confiado a mim o conteúdo da revista como editor-chefe. Através da minha atuação na revista eu pude aprender muita coisa e me manter sempre atualizado durante os últimos anos. Através dos artigos que escrevi ganhei experiência e tomei gosto pela escrita de conteúdo técnico. Posso afirmar com certeza que a semente desse livro foi plantada em alguns de meus artigos durante os últimos anos. Espero que essa parceria ainda dure bastante tempo!

Agradeço também a comunidade de padrões, tanto nacional quanto internacional, por ter me influenciado com toda sua cultura e ideias. Sou grato por terem me recebido de braços abertos e por terem fornecido um forum onde foi possível debater as minhas ideias e obter feedback dos trabalhos que estou realizando. Agra-

deço em especial pelo apoio e incentivo de Fabio Kon, Fábio Silveira, Jerffeson Souza, Uirá Kulezsa, Ademar Aguiar e Filipe Correia. Também agradeço pelos meus gurus Joseph Yoder e Rebecca Wirfs-Brock, com quem aprendi demais e tive discussões muito interessantes sobre modelagem e arquitetura de software.

Agradeço ao INPE, instituição onde acabei de ingressar, por ter me recebido de braços abertos e pela confiança que tem depositado em meu trabalho. Espero poder dar muitas contribuições e desenvolver diversos trabalhos interessantes durante os anos que estão por vir!

Finalmente agradeço a Casa do Código pelo convite para escrita desse livro. Apesar da vontade sempre ter existido, o convite de vocês foi o estopim para que esse projeto se tornasse algo concreto. Deixo um agradecimento em especial ao editor desse livro, Paulo Silveira, pelos questionamentos e sugestões que contribuiram de forma definitiva para o conteúdo desse livro.

Minibiografia do Autor

Eduardo Martins Guerra nasceu em Juiz de Fora em 1980 e cursou o ensino básico e médio em escola pública, o Colégio de Aplicação João XXIII. Desde essa época ele já despertou seu interesse pela programação, começando com a digitação de código Basic que vinha em revistas no seu defasado computador Exato Pro. A diversão era mais ver o efeito de pequenas modificações no código do que o jogo que saia como resultado. Nessa época, pouco depois, também fez um curso de Clipper, onde desenvolveu seu primeiro software: um gerenciador de canil!

Incentivado por seus professores, pela facilidade com disciplinas na área de exatas, prestou o vestibular para o ITA em 1998, logo após o término de seus estudos, e foi aprovado. Durante o curso, ele optou pela carreira militar e pelo curso de computação, além de participar de atividades extracurriculares como projetos para a empresa júnior e aulas no curso pré-vestibular para pessoas carentes, CASD Vestibulares. Nesses cinco anos, Eduardo se apaixonou pela área de desenvolvimento de software e pela possibilidade de usar constantemente sua criatividade para propor sempre soluções inteligentes e inovadoras. Dois dias depois de se formar, casou-se com sua mais forte paixão, sua esposa atual, Roberta.

Após formado, em 2003, foi alocado no Centro de Computação da Aeronáutica de São José dos Campos (CCA-SJ), onde ficaria pelos próximos 4 anos. Durante esse período, trabalhou com o desenvolvimento de software operacional para atender as necessidades da Força Aérea Brasileira, adquirindo uma valiosa experiência na área.

No final de 2005, com dedicação em tempo parcial, Eduardo conseguiu seu título de mestre em Engenharia da Computação e Eletrônica apresentando a dissertação “Um Estudo sobre Refatoração de Código de Teste”. Nesse trabalho, Eduardo desenvolve sua paixão pelo design de software, realizando um estudo inovador sobre boas práticas ao se codificar testes automatizados. Essa dissertação talvez tenha sido um dos primeiros trabalhos acadêmicos no Brasil no contexto de desenvolvimento ágil. No ano de conclusão de seu mestrado, veio sua primeira filha, Maria Eduarda.

Devido sua sede de conhecimento, ainda nesse período, estudou por conta própria e tirou sete certificações referentes a tecnologia Java, sendo na época um dos profissionais brasileiros com maior número de certificações na área. Além disso, ocupa o cargo de editor-chefe da revista MundoJ desde 2006, na qual já publicou dezenas de artigos técnicos de reconhecida qualidade. Esses fatos lhe deram uma visão da indústria de desenvolvimento de software que foi muito importante em sua trajetória, direcionando sua pesquisa e seus trabalhos para problemas reais e relevantes.

No CCA-SJ, Eduardo atuou de forma decisiva no desenvolvimento de frameworks. Seus frameworks simplificaram a criação das aplicações e deram maior produtividade para a equipe de implementação. Um deles, o SwingBean, foi transformado em um projeto open-source e já possui mais de 5000 downloads. A grande inovação e diferencial de seus frameworks estava no fato de serem baseados em metadados. A partir desse caso de sucesso, ele decidiu estudar mais sobre como a utilização de metadados poderia ser feita em outros contextos. Foi uma surpresa descobrir que, apesar de outros frameworks líderes de mercado utilizarem essas técnicas, ainda não havia nenhum estudo sobre o assunto. Foi, então, que Eduardo começou sua jornada para estudar e tornar acessível o uso dessa técnica por outros desenvolvedores, pois ele sabia do potencial que os frameworks baseados em metadados tem para agilizar o desenvolvimento de software e o tornar mais flexível.

Então, em 2007, ele ingressou no curso de doutorado do ITA pelo Programa de Pós-Graduação em Aplicações Operacionais – PPGAO. A pesquisa sobre frameworks baseados em metadados se encaixava perfeitamente no objetivo do programa. A criação de arquiteturas flexíveis, nas quais se pode acrescentar funcionalidade de forma mais fácil, é algo crítico para aplicações de comando e controle, foco de uma das áreas do programa. Orientado pelo professor Clovis Torres Fernandes, começou uma pesquisa que envolveu a análise de frameworks existentes, a abstração de técnicas e práticas, a criação de novos frameworks de código aberto e a execução de experimentos para avaliar os conceitos propostos.

Foi nessa época que Eduardo começou a participar e se envolveu na comunidade de padrões. Em 2008 submeteu para o SugarLoafPLoP em Fortaleza os primeiros padrões que identificou em frameworks que utilizavam metadados. Nesse momento ele se emplogou com o espírito de colaboração da comunidade de padrões, onde recebeu um imenso feedback do trabalho que estava realizando. Desde então se tornou um membro ativo da comunidade, publicando artigos com novos padrões em eventos no Brasil e no exterior. Além disso, em 2011 participou da organização do

MiniPLoP Brasil, em 2012 foi o primeiro brasileiro a ser chair da principal conferência internacional de padrões, o PLoP, e em 2013 está também participando também da organização do próximo MiniPLoP.

Enquanto realizava seu doutorado, Eduardo foi incorporado, ainda como militar, no corpo docente do Departamento de Ciência da Computação do ITA, onde pôde desenvolver uma nova paixão: ensinar! Durante esse período, ministrou aulas na graduação e em cursos de especialização, e orientou uma série de trabalhos que, de certa forma, complementavam e exploravam outros aspectos do que estava desenvolvendo em sua tese. Em consequência do bom trabalho realizado, foi convidado por três vezes consecutivas para ser o professor homenageado da turma de graduação Engenharia da Computação e duas vezes para a turma de especialização em Engenharia de Software.

Em 2010, apresentou seu doutorado chamado "A Conceptual Model for Metadata-based Frameworks e concluiu com sucesso essa jornada que deixou diversas contribuições. Devido a relevância do tema, foi incentivado pelos membros da banca a continuar os estudos que vem realizando nessa área. Apesar da tese ter trazido grandes avanços, ele tem consciência que ainda há muito a ser feito. Uma de suas iniciativas nessa área foi o projeto Esfinge (http://esfinge.sf.net) , que é um projeto guarda-chuva para a criação de diversos frameworks com essa abordagem baseada em metadados. Até o momento já existem três frameworks disponíveis e vários artigos científicos em cima de inovações realizadas nesses projetos. No mesmo ano que terminou seu doutorado, veio sua segunda filhinha, a Ana Beatriz.

Em 2012, Eduardo prestou concurso para o Instituto Nacional de Pesquisas Espaciais, onde assumiu o cargo de pesquisador no início 2013. Hoje ele segue com sua pesquisa na área de arquitetura, testes e design de software, buscando aplicar as técnicas que desenvolve para projetos na área especial. Adicionalmente, atua como docente na pós-graduação de Computação Aplicada desse instituto, onde busca passar o conhecimento que adquiriu e orientar alunos para contribuirem com essas áreas.

Porque mais um livro de padrões?

O primeiro livro de padrões "Design Patterns: Elements of Reusable Object-Oriented Software", conhecido como Gang of Four (GoF), já foi lançado a mais de 15 anos e de forma alguma seu conteúdo está ultrapassado. Em minha opinião, esse livro foi algo muito a frente de seu tempo, sendo que muitas de suas práticas só foram ser utilizadas de forma efetiva pela indústria alguns anos depois. Durante esses anos, diversos livros sobre esses padrões foram escritos, apresentando sua implementação em outras linguagens ou ensinando-os de forma mais didática. Desse contexto podem surgir as seguintes perguntas: será que é preciso mais um livro sobre padrões? O que esse livro trás de diferente?

Durante esses últimos anos tive diversos tipos de experiência que me fizerem ter diferentes visões sobre os padrões. Dentre essas experiências eu posso citar o ensino de modelagem de software e padrões, a utilização de padrões para o desenvolvimento de frameworks e aplicações, a identificação de novos padrões a partir do estudo de implementações existentes e discussões na comunidade sobre padrões e sua aplicabilidade. A partir disso acredito que tive a oportunidade de ter uma visão diferenciada sobre esses padrões, e é essa visão que procuro passar nesse livro. As seções a seguir descrevem alguns diferenciais desse livro em relação a outros existentes.

Uma Sequência Didática para Aprendizado

Percebi que muitos cursos sobre padrões, sendo eles dados por universidades ou empresas de treinamento, vão ensinando os padrões um por um. Mas qual é a melhor ordem para o ensino e a assimilação dos padrões? Isso era algo que era decidido por cada professor. Uma ordem inadequada nesse aprendizado pode impedir que o aluno não compreenda os princípios por trás da solução do padrão, o que dificulta sua compreenção a respeito de sua aplicabilidade e suas consequências. Isso também pode causar uma visão limitada da solução do padrão, o que impede sua adaptação para outros contextos similares.

Esse livro procura organizar os padrões em uma sequência didática para o aprendizado. Cada capítulo agrupa os padrões pelo tipo de técnica que utilizam em sua solução ou pelo tipo de problema que resolvem. Dessa forma, ao ver uma mesma técnica ser utilizada em padrões diferentes é possível compreender melhor a sua dinâmica e de que formas diferentes ela pode ser utilizada. Em capítulos que focam em tipos de problema, a visão das diferentes alternativas de solução e das suas diferenças, cria um senso crítico a respeito das possíveis alternativas de modelagem e quais os critérios que devem ser considerados para sua escolha.

Isso faz com que esse livro vá além da apresentação dos padrões e seja na verdade a respeito de técnicas para modelagem orientada a objetos. Os padrões são utilizados como uma referência para a discussão de cada uma dessas técnicas. Dessa forma, esse livro é recomendado para utilização como material base em cursos de graduação ou pós-graduação a respeito de técnicas de programação orientada a objetos ou programação orientada a objetos avançada.

Relação entre Padrões

Outro problema que percebo é que os padrões tendem a ser olhados de forma individual. Isso é uma consequência do formato que os padrões possuem e do fato de serem auto-contidos, ou seja, toda informação sobre o padrão estar contida em sua descrição. Isso por um lado é bom pois permite que alguém obtenha todas as informações sobre um determinado padrão em apenas um local. Porém, por outro lado, perde-se um pouco a ideia a respeito do relacionamento entre os padrões. Apesar de haver uma seção que descreve os padrões relacionados, ainda é uma descrição pequena para uma questão de tamanha importância.

Apresentar o relacionamento entre os padrões foi um dos objetivos desse livro. Apesar de cada padrão apresentar uma solução independente, é da combinação entre eles que é possível criar uma sinergia para criação de soluções ainda melhores. Dessa forma, a medida que o livro vai seguindo, e novos padrões vão sendo apresentados, existe uma preocupação em mostrar como eles podem ser combinados com os padrões anteriores ou como eles podem ser utilizados como uma alternativa a um outro padrão. Esse tipo de discussão é importante, pois além de saber a solução do padrão, também é essencial saber quando aplica-la.

Refatoração para Padrões

A modelagem de uma aplicação é uma questão dinâmica que evolui a medida

que o desenvolvimento do software vai seguindo seu curso. Apesar de ainda ser comum a modelagem das classes da aplicação a partir de diagramas antes do inicio da implementações, muitas vezes os problemas aparecem somente depois. Sendo assim, tão importante quanto saber como utilizar os padrões na modelagem inicial, é saber como refatorar um código existente para uma solução que utilize um determinado padrão. A refatoração constante do código é hoje uma das bases para a manutenção de sua qualidade ao longo do projeto.

Dessa forma, além de apresentar os padrões, esse livro também se preocupa em mostrar quais os passos que precisariam ser feitos para refatorar um código existente em direção a um determinado padrão. Este assunto, que já foi tema de um livro dedicado somente a ele, aqui é apresentado de forma fluida juntamente com os padrões. Isso é importante para que o leitor possa ter uma visão, não apenas estática em relação a utização de um padrão, mas de como um código existente pode evoluir e ser melhorado a partir de sua introdução a partir de pequenos passos.

Exemplos Realistas e do Dia-a-dia

Por mais que diagramas sejam úteis para que se tenha uma visão geral de uma solução, os desenvolvedores ainda tem muito a cultura do "*show me the code*", querendo ver na prática como um padrão pode ser implementado. Por mais que o GoF tenha sido um livro a frente de seu tempo, ele utiliza exemplos que hoje estão distantes de um dessenvolver comum, como a criação de uma suite de componentes gráficos ou um editor de texto. Alguns outros livros acabam trazendo "*toy examples*", que seriam exemplos simples e didáticos, mas que são fora do contexto de aplicações reais.

Acho muito importante que o leitor se identifique com os exemplos e faça uma ligação com o tipo de software que desenvolve. Ao se ver próximo aos exemplos e possuir uma experiência anterior com aquele tipo de problema, é muito mais fácil compreender as questões que estão envolvidas e as alternativas de solução com suas respectivas consequências. Nesse livro, busquei trabalhar com exemplos recorrentes em aplicações desenvolvidas por grande parte dos programadores, como geração de relatórios, carrinho de compras em aplicações de e-commerce e geração de arquivos para integração entre aplicações. Acredito que assim ficará bem mais claro como cada um poderá utilizar esses padrões em seu dia-a-dia.

Dicas e Referências para Implementações em Java

Muitos padrões apresentados por esse livro são amplamente utilizados em fra-

meworks e APIs Java. Muitas vezes, apenas por seguir as regras de uma determinada API, sem saber você já utilizando um determinado padrão. A flexibilidade que aquele framework consegue para ser instanciado na sua aplicação, muitas vezes é conseguido justamente pelo uso do padrão! O conhecimento de uma situação em que aquele padrão foi utilizada e, muitas vezes sem saber, você acabou o utilizando, também ajuda muito em sua compreenção.

Dessa forma, esse livro também cita diversas classes de APIs padrão da linguagem Java e de frameworks amplamente utilizados pela comunidade de desenvolvimento. Isso vai permitir que o desenvolvedor possa compreender melhor aquela solução inteligente utilizada naquele contexto, e trazer a mesma ideia para questões do seu próprio código.

Além disso, como o próprio nome do livro diz, os padrões de projeto são apresentados na linguagem Java e, dessa forma, acabam trazendo algumas dicas mais específicas da linguagemj para sua implementação. Essas dicas vão desde a utilização dos próprios recursos da linguagem, como a utilização de sua biblioteca de classes e, até mesmo, de frameworks externos.

Lista de Discussão

O design de software é um tema pelo qual me apaixonei justamente por ser algo em que o uso da criatividade é essancial, e cada apli cação tem suas particularidades, o que sempre traz novas questões a serem discutidas. O mesmo ocorre com padrões! Eles já estão aí a muitos anos e até hoje ainda existe muita discussão sobre eles! Sendo assim, eu gostaria que esse livro não fosse apenas uma comunicação de mão única onde eu escrevo e vocês leem, mas o início de um canal de comunicação para fomentar discussões e conversas sobre padrões e design de software em geral.

Sendo assim, criei uma lista de discussão com o nome **"Design Patterns em Java: Projeto orientado a objetos guiado por padrões"** no endereço **`projeto-oo-guiado-por-padroes@googlegroups.com`**.

Se você quer discutir, colocar suas dúvidas e saber eventos e novidades no mundo dos padrões, deixo aqui o meu convite para a participação no grupo!

Prefácio

Por Joseph Yoder

Já fazem cerca de 20 anos que o **Gang of Four** escreveu o livro inicial de padrões chamado **"Design Patterns: Elements of Reusable Object-Oriented Software"**. Desde aquela época, os padrões de projeto se tornaram muito conhecidos e uma parte essencial de uma boa modelagem em sistemas orientados a objetos. Adicionalmente, os padrões ganharam uma grande aceitação na comunidade de desenvolvimento de software, o que pode ser observado pela existência de diversas publicações e casos de sucesso. Existem também diversas conferências ao redor do mundo criadas aos moldes da primeira conferência sobre padrões, a **Pattern Languages of Programs** (**PLoP**).

Mesmo nos anos iniciais do Java, é possível observar como os padrões influenciaram a linguagem. Algumas dessas influências podem ser vistas claramente nas primeiras APIs do Java, como as interfaces `Iterator` e `Observer`, enquanto outras foram implementadas como parte da evolução dessas APIs, como a implementação dos **listeners**, que são uma implementação mais atual do padrão `Observer`.

Porém, é importante notar que um bom design não acontece por acidente, pois ele exige trabalho duro e evolução. De forma complementar, também não quer dizer que usando um padrão necessariamente se tem um bom design. Por exemplo, a linguagem Java foi lançada apenas alguns anos depois da publicação do primeiro livro de design patterns. Por causa disso, muitas das primeiras bibliotecas da linguagem foram influenciadas pelo livro e muitos padrões foram incorporados no design de suas principais bibliotecas e frameworks. No entanto, isso nem sempre guiou as soluções para o melhor design e foram necessárias várias evoluções em suas classes depois que diversos problemas foram encontrados nas primeiras versões.

Um bom exemplo pode ser visto no tratamento de eventos do Java AWT. No AWT 1.0, o tratamento de eventos utilizava o padrão **`Chain of Responsibility`**. A princípio essa parecia uma solução adequada ao olhar para os

requisitos. O problema com essa implementação era que precisava-se da utilização de um **`Mediator`** ou da criação de diversas subclasses. Isso poderia causar problemas de desempenho, visto que para o evento ser tratado era necessário percorrer toda a cadeia para descobrir qual era a classe que deveria processa-lo. Isso muitas vezes direcionava o desenvolvedor a criação de uma solução muito complexa devido ao grande número de subclasses que precisava criar.

A partir disso, o AWT 1.1 passou a tratar os eventos de forma mais eficente utilizando os padrões **`Observer`** e **`Adapter`**. Essa modelagem evoluiu depois para a utilização de listeners. Essa acabou sendo uma solução muito mais limpa e eficiente, visto que apenas as classes interessadas no evento eram notificadas quando ele acontecia. Essa história claramente mostra que o design de um software deve levar em consideração as consequências positivas e negativas de cada decisão, e só porque ele utiliza padrões não quer dizer que aquela é a melhor alternativa de design. Veja que isso não quer dizer que utilizar padrões leva a decisões erradas de design, mas que, pelo contrário, a utilização de um outro padrão mais adequado se mostrou uma solução melhor.

Apenas saber **como** aplicar os padrões não é o suficiente para um bom design. Entender quando e onde aplica-los é tão importante quanto, senão mais importante. De fato, no decorrer do tempo, mesmo um bom design inicial pode ser comprometido por sucessivas revisões arquiteturais. Em 1998, foi feita a afirmação que a arquitetura que realmente predominava na prática era a **`Big Ball of Mud`** (traduzindo Grande Bola de Lama). E mesmo com muito trabalho e dedicação ainda é possível acabar com um **`Big Ball of Mud`** se você não está comprometido a manter o código sempre limpo.

Existem muitas forças e bons motivos que podem levar a uma arquitetura extremamente complexa. De fato, arquitetos e times de desenvolvimento experientes estão constantemente fazendo exatamente o que deveria ser feito quando se deparam com "lama" e complexidade desnecessária em seu sistema.

Quando se tenta manter uma arquitetura, é importante tentar proteger certas partes do design. Grandes sistemas possuem partes mudam em diferentes velocidades. Existem certas ações e padrões que você pode utilizar para isolar e definir divisões em volta dessas diferentes partes, tanto para tornar a arquitetura estável, quanto para possibilitar as mudanças onde elas são necessárias. De fato, essa é uma premissa importante dos padrões de projeto. Isso se torna uma consideração ainda mais importante quando evoluimos a estrutura das aplicações para um mundo onde parte delas está localizada na nuvem.

Eu conheço o autor desse livro (Eduardo) a muitos anos. Eduardo tem grande experiência em padrões, design orientado a objetos e Java. Eu já colaborei e trabalhei com ele em projetos orientados a objetos e publicações, incluindo a implementação de diversos padrões de projeto e a construção de frameworks. Eduardo tem muita experiência na construção de frameworks, atividade que exige um conhecimento profundo de padrões, além de quando e onde utiliza-los. Ele tem mais do que uma compreenção de "como" implementar um padrão em Java. Eduardo claramente compreende o que está em jogo em um bom design orientado a objetos e como utilizar os padrões para um design mais limpo, mais reutilizável e mais fácil de mudar.

Qualquer desenvolvedor para se tornar proficiente em um design orientado a objetos precisará compreender não apenas o básico sobre padrões, mas também principios mais avançados a respeito de suas consequências e o contexto em que devem ser utilizados. Este livro é mais do que um livro de receitas sobre como aplicar os padrões, pois, pelo contrário, ele traz também importantes princípios e se aprofunda nessas técnicas avançadas de design. O leitor irá descobrir nesse livro um guia eficiente a respeito de como aplicar os padrões, como escolher o padrão mais adequado e como saber quando um design deve ser refatorado para a introdução de um novo padrão.

Após ler esse livro, é possível continuar essa jornada sobre o conhecimento de padrões através de conferências sobre o assunto, como o SugarLoafPLoP no Brasil. Este assunto tem se tornado de extrema importância para todas as facetas dos sistemas de software, principalmente aqueles que estão em constante evolução para se manterem atualizados frente a novos requisitos.

Sumário

Versão: 16.2.5

Capítulo 1

Entendendo Padrões de Projeto

"Eu diria algo como 'Software está gerenciando o mundo'. Nosso trabalho é apenas polinizá-lo ..."
– Brian Foote

Imagine que uma pessoa tenha aprendido diversas técnicas de pintura. A partir desse conhecimento ela saberá como pegar um pincel, como misturar as cores e como trabalhar com diferentes tipos de tinta. Será que somente com esse conhecimento ela conseguirá pintar um quadro? Note que ela sabe tudo que se precisa para realizar a pintura, porém todo esse conhecimento não é válido se a pessoa não souber como utilizá-lo. Para realizar essa tarefa, é necessário, além de conhecimento, ter habilidade, que é algo que só se aprende com muita prática e treino. Saber as técnicas é apenas o primeiro passo...

Com programação acontece um fenômeno similar. Quando se aprende uma linguagem orientada a objetos e seus recursos, isso é equivalente a se aprender a pegar em um pincel. Saber, por exemplo, como utilizar herança e polimorfismo não é o suficiente para distinguir em quais situações eles devem ser empregados de forma

apropriada. É preciso conhecer os problemas que podem aparecer durante a modelagem de um sistema e saber quais as soluções que podem ser implementadas para equilibrar requisitos, muitas vezes contraditórios, com os quais se está lidando.

Felizmente existe uma forma de conseguir esse conhecimento sem precisar passar pelo caminho tortuoso de errar várias vezes antes de aprender a forma correta. É um prazer poder apresentar nesse livro os *Design Patterns*, padrões de projeto na tradução mais utilizada, uma forma de se documentar uma solução para um problema de modelagem. Mais do que isso, padrões não são novas soluções, mas soluções que foram implementadas com sucesso de forma recorrente em diferentes contextos. A partir deles, é possível aprender com a experiência de outros desenvolvedores e absorver esse conhecimento que eles levaram diversos anos para consolidar.

O objetivo desse capítulo é fazer uma revisão dos principais conceitos da orientação a objetos e apresentar como os padrões de projeto podem ser utilizados para adquirir habilidade em modelagem de software. Ao seu final, será apresentada uma visão geral de como esses padrões serão apresentados no decorrer do livro.

1.1 Conceitos da Orientação a Objetos

Em linguagens mais antigas, não havia nenhuma separação dos elementos de código. Ele era criado em único bloco, no qual para se executar desvios no fluxo de execução, como para a criação de loops e condicionais, era preciso realizar saltos através do temido comando `goto`. Isso tornava o código muito difícil de ser gerenciado e reutilizado, complicando seu desenvolvimento e manutenção. Foi dessa época que surgiu o termo *"código macarrônico*", que se referia à dificuldade de compreensão do fluxo de execução desses programas.

Em seguida, para resolver esse problema, foi criada a **programação estruturada**. Nesse novo paradigma surgiram estruturas de controle que permitiam a criação de comando condicionais, como o `if` e o `switch`, e comandos iterativos, como o `for` e o `while`. Uma outra adição importante foram as funções. A partir delas era possível dividir a lógica do programa, permitindo sua modularização. Isso facilita que essas funções possam ser reutilizadas em outras aplicações. A maior dificuldade nesse paradigma era como lidar com os dados e o comportamento associado a eles. Era comum a criação de variáveis globais, que facilmente poderiam acabar causando problemas por estarem com valores inconsistentes. Outro problema ocorria com dados relacionados que podiam ser livremente modificados e facilmente ficarem inconsistentes.

Como uma evolução da programação estruturada surgiu a **programação orientada a objetos**. Além da estruturação da lógica, com esse paradigma era possível a estruturação dos dados e conceitos do software. Sendo assim, pode-se colocar toda lógica relacionada a um conjunto de dados junto com ele. Além disso, a partir da possibilidade de abstração de conceitos consegue-se desacoplar componentes, obtendo um maior reúso e uma maior flexibilidade.

Os tópicos a seguir exploram com maior detalhe os principais conceitos da programação orientada a objetos.

Classes e Objetos

Na programação orientada a objetos são definidos novos tipos através da criação de **classes**, e esses tipos podem ser instanciados criando **objetos**. A ideia é que um objeto represente uma entidade concreta enquanto sua classe representa uma abstração dos seus conceitos. Se, por exemplo, tivermos uma classe `Gato`, o Garfield e o Frajola seriam objetos dessa classe. Adicionalmente, uma classe `CarteiraDeIdentidade`, que representa uma abstração, teria como instâncias a minha e a sua carteira de identidade.

Eu vejo muitas pessoas utilizarem a metáfora *"a classe seria a forma e o objeto seria o pão"* para explicar a relação entre classes e objetos. Pessoalmente, não gosto dessa analogia pois ela leva a entender que a classe é o que produz o objeto, o que seria mais próximo do conceito de fábrica (que será visto no Capítulo 6 desse livro). É importante perceber que os objetos representam entidades concretas do seu sistema, enquanto as classes abstraem suas características.

A classe possui estado e comportamento, que são representados respectivamente pelos atributos e métodos definidos. Enquanto uma classe possui uma característica, um objeto possui um valor para aquela característica. Por exemplo, um `Gato` possui a cor do pelo, enquanto o Garfield possui seu pelo laranjado. A classe possui um comportamento e o objeto pode realizar aquele comportamento. Seguindo o mesmo exemplo, enquanto um `Gato` pode correr, o Frajola realmente corre para tentar pegar o Piu-piu.

Projetar um sistema orientado a objetos consiste em definir suas classes e como elas colaboram para conseguir implementar os requisitos de uma aplicação. É importante ressaltar que não é só porque uma linguagem orientada a objetos está sendo utilizada que se estará modelando de forma orientada a objetos. Se suas classes não representam abstrações do sistema e são apenas repositórios de funções, você na verdade está programando segundo o paradigma estruturado.

Herança

Se uma aplicação precisa de um conjunto de objetos, devem haver classes que abstraiam esses conceitos. Porém, esses mesmos objetos podem precisar ser tratados em partes diferentes do software a partir de diferentes níveis de abstração. O Garfield e o Frajola podem ser tratados como gatos em uma parte do sistema, porém outra, que precisar também lidar com os objetos Pica-pau e Mickey Mouse, pode precisar de um conceito mais abstrato, como animal ou personagem.

A **herança** é uma característica do paradigma orientado a objetos que permite que abstrações possam ser definidas em diversos níveis. Considerando que você tem uma classe, ela pode ser especializada por uma outra que irá definir um conceito mais concreto. Por outro lado, um conjunto de classes pode ser generalizado por uma classe que representa um conceito mais abstrato. Quando uma classe estende outra, ela não só herda a estrutura de dados e o comportamento da superclasse, mas também o contrato que ela mantém com os seus clientes.

Um dos grandes desafios da modelagem orientada objetos é identificar os pontos em que essa abstração deve ser utilizada e que será aproveitada de forma adequada pelo sistema. Pelo fato de um sistema de software ser uma representação de algum processo que acontece no mundo real, isso não significa que todas as abstrações existentes precisam ser representadas no software. Um avião, por exemplo, seria representado de forma completamente diferente em um simulador de voo, em um sistema de controle de tráfego aéreo e no sistema de vendas de uma companhia aérea. Saber qual a representação e as abstrações mais adequadas é um grande desafio.

Encapsulamento

Sempre que falo sobre encapsulamento, inicio com a frase *"A ignorância é uma benção!"*. Não me entendam mal, mas com a complexidade das coisas com que lidamos nos dias de hoje, é realmente muito bom não precisar saber como elas funcionam para utilizá-las. Fico feliz em poder assistir minha televisão, mudar de canal, aumentar o volume e não ter a menor ideia de como essas coisas estão funcionando. Imagine como seria complicado poder utilizar novas tecnologias se elas exigissem uma compreensão profunda de seu funcionamento interno.

Essa mesma questão acontece com software. Antes da programação estruturada, o desenvolvedor precisava conhecer os detalhes de implementação de cada parte do código. Com a criação das funções, houve um certo avanço em relação à divisão do código, porém a utilização de algumas funções ainda demandava um conheci-

mento sobre seu funcionamento interno, como que variáveis elas estão acessando e atualizando.

O **encapsulamento** é um conceito importante da orientação a objetos que diz que deve haver uma separação entre o comportamento interno de uma classe com a interface que ela disponibiliza para os seus clientes. Isso permite que o desenvolvedor que utilizar uma classe precise saber somente como interagir com ela, abstraindo seu comportamento interno. Este é um conceito muito poderoso que possibilita o desacoplamento entre as classes, podendo ser utilizado para uma melhor divisão do software em módulos.

Os métodos *getters* e *setters*, que são utilizados respectivamente para acessar e modificar propriedades em um objeto, são um exemplo do uso do encapsulamento. Para os clientes da classe, apenas uma informação está sendo recuperada ou modificada, porém a implementação por trás pode realizar outras coisas. Ao recuperar uma informação, ela pode ser calculada a partir de outros dados, e ao modificar uma informação, pode haver uma validação dos dados. É importante lembrar que o uso desses métodos de acesso é apenas um começo para o encapsulamento de uma classe. A estrutura de dados pode ser utilizada e manipulada também por outros métodos de forma totalmente transparente à classe cliente.

A linguagem Java provê as interfaces como um recurso de linguagem, que permite que a definição do contrato externo da classe possa ser feita de forma separada da implementação. Quando uma classe implementa uma interface é como se estivesse assinando um documento, no qual ela se compromete a implementar seus métodos. Dessa forma, os clientes dela não precisam conhecer a classe onde está a implementação, mas apenas a interface. Adicionalmente, diferentes implementações podem ser utilizadas pelo cliente sem a modificação de seu código.

Interface ou classe abstrata?

Várias vezes já me fizeram a pergunta: *"quando devo utilizar uma classe abstrata e quando devo utilizar uma interface?"*. Tanto as classes abstratas quanto as interfaces podem definir métodos abstratos que precisam ser implementados pelas classes que respectivamente a estende ou implementa. Porém apenas as classes abstratas podem possuir métodos concretos e atributos. Apesar dessa diferença, a resposta para pergunta é mais conceitual do que relacionada com questões de linguagem.

Quando a abstração que precisar ser criada for um conceito, ou seja, algo que possa ser refinado e espacializado, deve-se utilizar uma classe abstrata. Quando a abstração é um comportamento, ou seja, algo que uma classe deve saber fazer, então a melhor solução é a criação de uma interface. Imagine um jogo no qual existem naves que se movem. Se sua abstração representa uma nave, então você está representando um conceito e deve utilizar uma classe abstrata. Por outro lado, se sua abstração representa algo que se move, então o que está sendo abstraído é um comportamento e a melhor solução é usar uma interface.

Polimorfismo

O **polimorfismo** é na verdade uma consequência da utilização de herança e da utilização de interfaces. Não faria sentido utilizar herança para criar novas abstrações se o objeto não pudesse ser visto como uma dessas abstrações. Equivalentemente, não faria sentido haver uma interface se a classe não pudesse ser tratada como algo que possui aquele comportamento. É através do polimorfismo que um objeto pode ser visto como qualquer uma de suas abstrações.

A palavra polimorfismo significa *"múltiplas formas"*, o que indica que um objeto pode assumir a forma de uma de suas abstrações. Em outras palavras, qualquer objeto pode ser atribuído para uma variável do tipo de uma de suas superclasses ou para o tipo de suas interfaces. Isso é um recurso extremamente poderoso, pois um código pode utilizar uma classe que não conhece, se souber trabalhar com uma de suas abstrações. Isso pode ter um impacto tremendo na reutilização e desacoplamento das classes.

Um bom exemplo para entender polimorfismo é a interface `Comparable` que é

provida pela API padrão da linguagem Java. Os algoritmos de ordenação presentes na classe `Collections` sabem ordenar listas de qualquer objeto que implementa essa interface, o que inclui classes como `Number`, `String` e `Date`. Dessa forma, se uma classe da sua aplicação implementar essa interface, então os algoritmos saberão ordenar uma lista de suas instâncias. O mais interessante é que esse algoritmo utiliza sua classe, mesmo tendo sido desenvolvido muito antes dela existir.

1.2 Mas esses conceitos não são suficientes?

Sem mais delongas, já irei responder que não são suficientes. Na verdade, compreender bem esses conceitos é apenas o primeiro passo para a realização de um projeto orientado a objetos. Entender o significado da herança e saber como implementá-la na linguagem de programação não é suficiente para saber em que situações o seu emprego é adequado. Saber como uma classe implementa uma interface não basta para saber quando seu uso irá ajudar a resolver um problema.

Realizar a modelagem de um software é um imenso jogo de raciocínio, como em um tabuleiro de xadrez. Da mesma forma que muitas vezes é preciso entregar uma peça para se atingir um objetivo, no projeto de um software é comum abrir mão de certas características para se obter outras. Por exemplo, ao se adicionar uma verificação de segurança, o desempenho pode ser degradado. Ou mesmo, ao se adicionar flexibilidade, a complexidade de uso da solução pode aumentar.

São raros os benefícios que vêm sem nenhuma consequência negativa. Durante o projeto de um software o desenvolvedor está sentado em uma mesa de negociação, e do outro lado estão sentados os requisitos que precisam ser atendidos. Ao colocar na mesa uma solução, devem ser avaliados as perdas e os ganhos obtidos para tentar equilibrar os atributos de qualidade do sistema. Muitas vezes é preciso combinar mais de uma solução para que o benefício de uma compense a desvantagem da outra.

Nesse jogo, as soluções que podem ser empregadas são as principais armas do desenvolvedor. O desenvolvedor com menor experiência conhece uma quantidade menor de soluções, o que muitas vezes o leva a adotar um projeto inadequado às necessidades do software. A elaboração de uma solução partindo do zero traz de forma intrínseca o risco da inovação, e muitas vezes acaba-se deixando de considerar outras alternativas.

Mas como conhecer diversas soluções sem precisar passar por vários anos alternando entre escolhas certas e erradas? Como saber o contexto em que essas soluções são adequadas e quais são as contrapartidas dos benefícios da solução? Felizmente

existe uma forma na qual desenvolvedores mais experientes expressam seu conhecimento em um determinado domínio. E então eu apresento os padrões de projeto, os *Design Patterns*!

1.3 O Primeiro Problema: Cálculo do Valor do Estacionamento

Para ilustrar como o livro vai proceder, essa seção vai apresentar um primeiro problema de projeto, a partir do qual serão exploradas as alternativas de solução. Apesar de grande parte dos problemas ser de sistemas fictícios e criados para ilustrar a aplicação dos padrões, as situações apresentadas por eles são realistas e aplicáveis em softwares reais.

Considere o sistema de um estacionamento que precisa utilizar diversos critérios para calcular o valor que deve ser cobrado de seus clientes. Para um veículo de passeio, o valor deve ser calculado como R$2,00 por hora, porém, caso o tempo seja maior do que 12 horas, será cobrada uma taxa diária, e caso o número de dias for maior que 15 dias, será cobrada uma mensalidade. Existem também regras diferentes para caminhões, que dependem do número de eixos e do valor da carga carregada, e para veículos para muitos passageiros, como ônibus e vans. O código a seguir apresenta um exemplo de como isto estava desenvolvido.

```
public class ContaEstacionamento {

    private Veiculo veiculo;
    private long inicio, fim;

    public double valorConta() {
        long atual = (fim==0) ? System.currentTimeMillis() : fim;
        long periodo = inicio - atual;
        if (veiculo instanceof Passeio) {
            if (periodo < 12 * HORA) {
                return 2.0 * Math.ceil(periodo / HORA);
            } else if (periodo > 12 * HORA && periodo < 15 * DIA) {
                return 26.0 * Math.ceil(periodo / DIA);
            } else {
                return 300.0 * Math.ceil(periodo / MES);
            }
        } else if (veiculo instanceof Carga) {
            // outras regras para veículos de carga
```

```
        }
        // outras regras para outros tipos de veículo
    }

}
```

Como é possível observar, o código utilizado para calcular os diversos condicionais é complicado de se entender. Apesar de apenas parte da implementação do método ter sido apresentada, pode-se perceber que a solução utilizada faz com que o método `valorConta()` seja bem grande. As instâncias da classe `ContaEstacionamento` relacionadas com os veículos que estão no momento estacionados são exibidas para o operador e têm seu tempo atualizado periodicamente.

Até então, o próprio desenvolvedor sabia que o código estava ruim, mas como o código estava funcionando, preferiu deixar da forma que está. Porém, o software começou a ser vendido para outras empresas e outras regras precisariam ser incluídas. Alguns municípios possuem leis específicas a respeito do intervalo de tempo para o qual um estacionamento definir sua tarifa. Além disso, diferentes empresas podem possuir diferentes critérios para cobrar o serviço de seus clientes.

A solução da forma como está não irá escalar para um número grande de regras! O código que já não estava bom poderia crescer de uma forma descontrolada e se tornar não gerenciável. O desenvolvedor sabia que precisaria refatorar esse código para uma solução diferente. O que ele poderia fazer?

Usando Herança

A primeira ideia que o desenvolvedor pensou foi na utilização de herança para tentar dividir a lógica. Sua ideia era criar uma superclasse em que o cálculo do valor da conta seria representado por um método abstrato, o qual seria implementado pelas subclasses com cada uma das regras. A Figura 1.1 apresenta como seria essa solução.

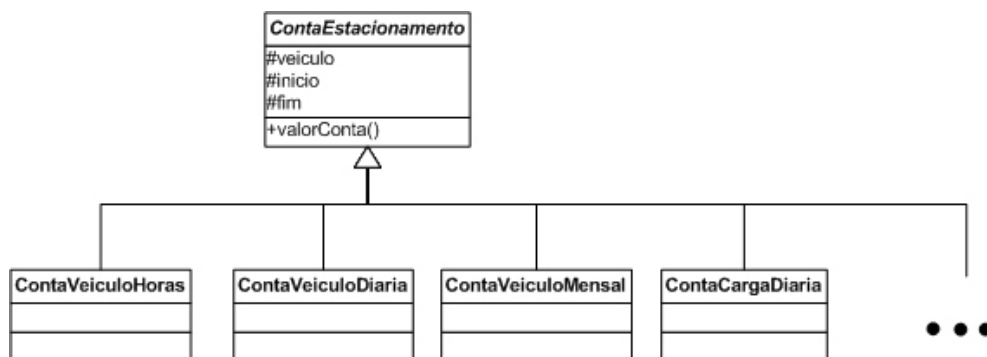



Figura 1.1: Utilização de herança para dividir a lógica

Um dos problemas dessa solução é a explosão de subclasses que vai acontecer, devido às várias possibilidades de implementação. Outra questão é que utilizando herança, depois não é possível alterar o comportamento uma vez que classe foi instanciada. Por exemplo, depois de 12 horas que um veículo estiver estacionado, o comportamento do cálculo da tarifa deve ser alterado da abordagem por hora para a abordagem por dia. Quando se cria o objeto como sendo de uma classe, para mudar o comportamento que ela implementa, é preciso criar uma nova instância de outra classe. Isso é algo indesejável, pois a mudança deveria acontecer dentro da própria classe!

A segunda solução que o desenvolvedor pensou foi utilizar a herança mas com uma granularidade diferente. Sua ideia era que cada subclasse possuísse o código relacionado a uma abordagem de cobrança para um tipo de veículo. Por exemplo, no caso acima, seria apenas uma subclasse para veículos de passeio e ela conteria os condicionais de tempo necessários. Dessa forma, a mesma instância seria capaz de fazer o cálculo.

Ao começar a seguir essa ideia, o desenvolvedor percebeu que nas subclasses criadas ocorria bastante duplicação de código. Um dos motivos é que a cada pequena diferença na abordagem, uma nova subclasse precisaria ser criada. Se a mudança fosse pequena, o resto do código comum precisaria ser duplicado. A Figura 1.2 ilustra essa questão. Uma das classes considera a tarifa de veículo de passeio como apresentado anteriormente e a outra considera que a primeira hora precisa ser dividida em períodos de 15 minutos (uma lei que existe no município de Juiz de Fora) e em seguida faz a cobrança por hora como mostrada anteriormente. Observe que se houvesse uma classe com os períodos de 15 minutos mais a cobrança por dia e por mês,

mais código ainda seria duplicado.

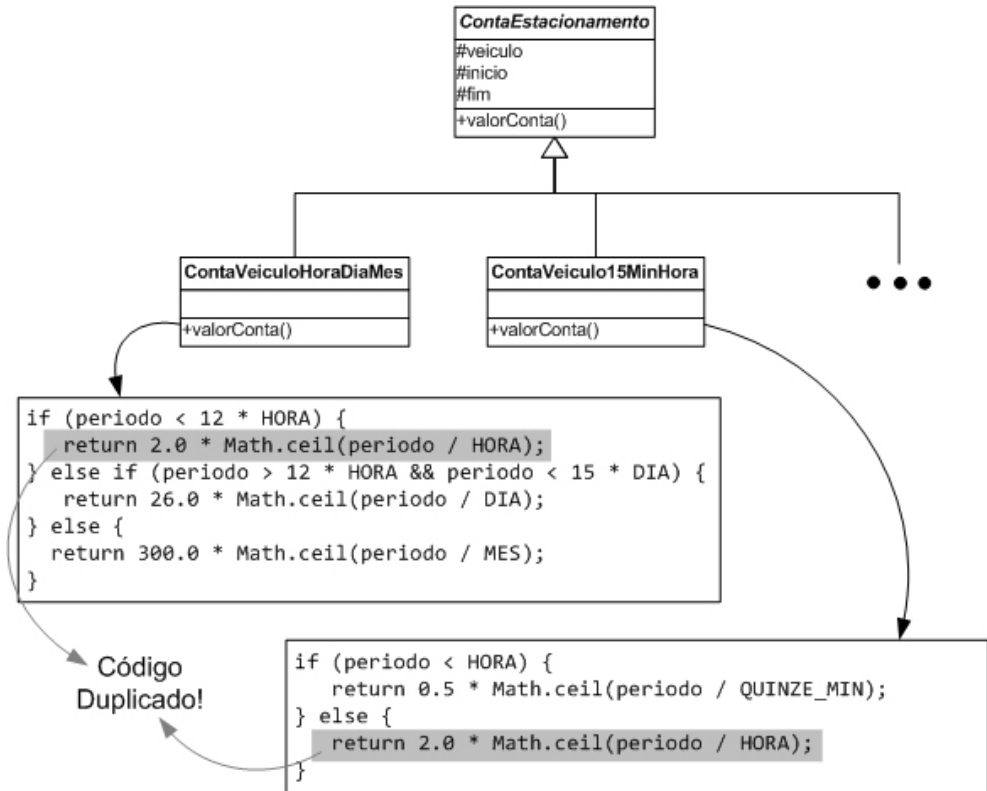



Figura 1.2: Duplicação de código com a implementação menos granular

Mas será que é possível ter uma forma de manter a mesma instância de `ContaEstacionamento` sem precisar duplicar código?

Pensando em Composição

Ao pensar melhor, o desenvolvedor decide que a herança não é a melhor abordagem para resolver o problema. Ele precisa de uma solução que permita que diferentes algoritmos de cálculo de tarifa possam ser utilizados pela classe. Adicionalmente, é desejável que não haja duplicação de código e que o mesmo algoritmo de cálculo possa ser utilizado para diferentes empresas. Além disso, uma classe deve poder iniciar a execução com um algoritmo e o mesmo ser trocado posteriormente.

Uma solução que se encaixa nos requisitos descritos é a classe

`ContaEstacionamento` delegar a lógica de cálculo para a instância de uma classe que a compõe. Dessa forma, para trocar o comportamento do cálculo do valor do estacionamento, basta trocar a classe que está o compondo. Essa classe também poderia ser parametrizada e ser reutilizada no contexto de diversas empresas. A figura 1.3 mostra o diagrama que representa essa solução.

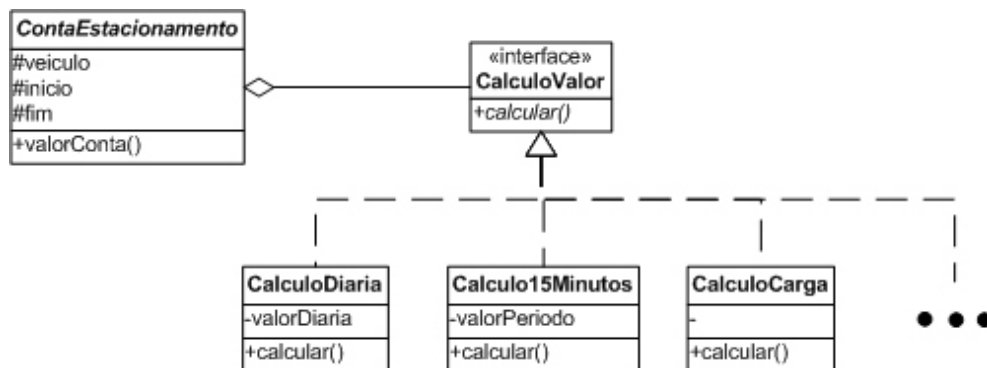



Figura 1.3: Utilizando a composição para permitir a troca do algoritmo

Agora apresentamos a classe `ContaEstacionamento` delegando para uma classe que a compõe o calculo do valor do estacionamento. A interface `CalculoValor` abstrai o algoritmo de cálculo da tarifa. Observe que existe um método que permite que o atributo `calculo` seja alterado, permitindo a mudança desse algoritmo depois que o objeto foi criado.

```
public class ContaEstacionamento {

    private CalculoValor calculo;

    private Veiculo veiculo;
    private long inicio;
    private long fim;

    public double valorConta() {
        return calculo.calcular(fim-inicio, veiculo);
    }

    public void setCalculo(CalculoValor calculo){
        this.calculo = calculo;
```

```
    }
}
```

A seguir, a classe `CalculoDiaria` mostra um exemplo de uma classe que faz o cálculo da tarifa por dia. Observe que essa classe possui um atributo que pode ser utilizado para parametrizar partes do algoritmo. Dessa forma, quando a estratégia for alterada para o calculo do valor por dia, basta inserir a instância dessa classe em `ContaEstacionamento`. Vale também ressaltar que essa mesma classe pode ser reaproveitada para diferentes empresas em diferentes momentos, evitando assim a duplicação de código.

Listagem 1.1 - Exemplo de classe que faz o cálculo da tarifa:

```
public class CalculoDiaria implements CalculoValor {

    private double valorDiaria;

    public CalculoDiaria(double valorDiaria){
        this.valorDiaria = valorDiaria;
    }

    public double calcular() {
        return valorDiaria * Math.ceil(periodo / HORA);
    }
}
```

Reconhecendo a Recorrência da Solução

Após implementar a solução, o desenvolvedor, orgulhoso de sua capacidade de modelagem, começa a pensar que essa mesma solução pode ser utilizada em outras situações. Quando houver um algoritmo que pode variar, essa é uma forma de permitir que novas implementações do algoritmo possam ser plugadas no software. Por exemplo o cálculo de impostos, como pode variar de acordo com a cidade, poderia utilizar uma solução parecida.

Na verdade, ele se lembrou que já tinha visto uma solução parecida em um sistema com que havia trabalhado. Como o algoritmo de criptografia que era utilizado para enviar um arquivo pela rede poderia ser trocado, existia uma abstração comum a todos eles que era utilizada para representá-los. Dessa forma, a classe que enviava o arquivo, era composta pela classe com o algoritmo. Sendo assim, o desenvolvedor decidiu conversar com o colega a respeito da coincidência da solução utilizada.

Sua surpresa foi quando ele disse que não era uma coincidência, pois ambos haviam utilizado o padrão de projeto `Strategy`.

Eu já usei isso e nem sabia que era um padrão!

Uma reação comum quando certas pessoas têm seu primeiro contato com padrões é ver que já utilizou aquela solução anteriormente. Isso é normal, pois o próprio nome *"padrão"* significa que aquela é uma solução que já foi utilizada com sucesso em diversos contextos. Nesse momento, algumas pessoas pensam: *Para que eu preciso saber os padrões se eu posso chegar a essas soluções sozinho?*.

Observe que para o desenvolvedor chegar a essa solução ele demorou um certo tempo. Mesmo assim, podem haver outras soluções que ele deixou de explorar e considerar, e podem haver consequências que muitas vezes não são levadas em conta. Ao aprender padrões, o desenvolvedor adquire o conhecimento não apenas da estrutura empregada, mas sobre o contexto em que ele é utilizado e as trocas feitas para se equilibrar os requisitos. Dessa forma, não se engane achando que os padrões não irão te acrescentar conhecimento, pois mesmo os desenvolvedores mais experientes aprendem muito com eles .

1.4 Strategy: o Primeiro Padrão!

"Por mais bonita que seja a estratégia, ocasionalmente deve-se olhar os resultados."
– - Sir Winston Churchill

O `Strategy` é um padrão que deve ser utilizado quando uma classe possuir diversos algoritmos que possam ser utilizados de forma intercambiável. A solução proposta pelo padrão consiste em delegar a execução do algoritmo para uma instância que compõe a classe principal. Dessa forma, quando a funcionalidade for invocada, no momento de execução do algoritmo, será invocado um método da instância que a compõe. A Figura 1.4 apresenta um diagrama que mostra a estrutura básica do padrão.

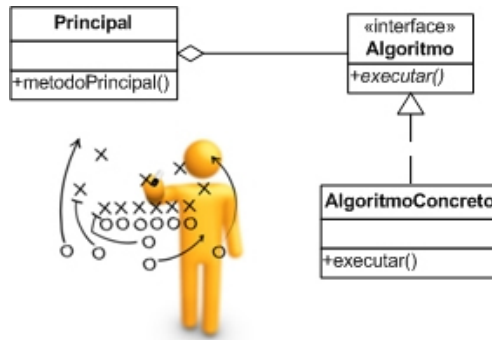



Figura 1.4: Estrutura do padrão Strategy

Uma das partes principais de um padrão diz respeito às suas consequências, pois é a partir delas que o desenvolvedor vai avaliar se essa é uma boa escolha ou não para seus requisitos. No caso do **`Strategy`**, a principal consequência positiva é justamente o fato de o algoritmo poder ser alterado sem a modificação da classe. A partir dessa estrutura, novas implementações dele podem ser criadas e introduzidas posteriormente.

Outro ponto positivo do padrão está no fato da lógica condicional na classe principal ser reduzida. Como a escolha do algoritmo está na implementação do objeto que está compondo a classe, isso elimina a necessidade de ter condicionais para selecionar a lógica a ser executada. Outra consequência positiva está no fato de a implementação poder ser trocada em tempo de execução, fazendo que o comportamento da classe possa ser trocado dinamicamente.

No mundo dos padrões, vale a expressão *"nem tudo são flores"*, e é importante conhecer as consequências negativas do padrão que está sendo utilizado. No caso do **`Strategy`**, isso acontece no aumento da complexidade na criação do objeto, pois a instância da dependência precisa ser criada e configurada. Caso o atributo seja nulo, a classe pode apresentar um comportamento inesperado. Outro problema dessa solução está no aumento do número de classes: há uma para cada algoritmo, criando uma maior dificuldade em seu gerenciamento.

1.5 O que são padrões?

Um padrão descreve um conjunto composto por um contexto, um problema e uma solução. Em outras palavras, pode-se descrever um padrão como uma solução para

um determinado problema em um contexto. Porém um padrão não descreve qualquer solução, mas uma solução que já tenha sido utilizada com sucesso em mais de um contexto. Exatamente por esse motivo que a descrição dos padrões normalmente sempre indica alguns de seus usos conhecidos. Um padrão não descreve soluções novas, mas soluções consolidadas!

A ideia dos padrões vem do trabalho de Christopher Alexander na área de arquitetura de cidades e construções [5]. Neste livro, cada padrão trazia uma solução adequada a um problema, que poderia ser reutilizada e adaptada para diversas situações. A disseminação dos padrões na comunidade de desenvolvimento de software iniciou-se com o conhecido livro *"Design Patterns: Elements of Reusable Object-Oriented Software*" [10]. Ele descrevia soluções de projeto orientado a objetos que são utilizadas até hoje por desenvolvedores de todo mundo. Esse livro também é conhecido como **GoF**, um acrônimo de *Gang of Four*, uma referência aos seus quatro autores.

Para ser um padrão, uma solução não basta ser recorrente, mas precisa ser uma boa solução (caso contrário é um anti-padrão!). Além disso, é comum que ela traga outras partes, como a estrutura da solução, como funciona sua dinâmica e as consequências positivas e negativas de sua aplicação. Outra parte importante é o relacionamento com os outros padrões, pois mostra outros que oferecem uma outra alternativa, ou padrões que podem complementar essa solução para compensar suas desvantagens.

Muitos desenvolvedores acham que o padrão se resume à sua estrutura, normalmente representada através de um diagrama de classes, e esquecem de todo o resto de sua descrição. Isso é um uso inadequado dos padrões, pois se a solução for utilizada no contexto errado, ela pode atrapalhar mais que ajudar. Se observarmos todos os padrões, é possível observar que a estrutura de alguns são bastante similares. Porém os problemas que essas estruturas vão resolver podem ser completamente diferentes. Por outro lado, a estrutura apresentada no padrão é apenas uma referência, pois é possível aplicá-lo com diversas adaptações estruturais de acordo com as necessidades da aplicação. É por esse motivo que existem fortes críticas a ferramentas possuem uma abordagem de gerar a estrutura de padrões de forma automática.

Padrões não se refletem em pedaços de código ou componentes que são reutilizados de forma igual em diversas aplicações, eles são um conhecimento que deve estar na cabeça dos desenvolvedores. A partir desse conhecimento, os desenvolvedores devem avaliar o contexto e o problema que querem resolver e adaptar, e combinar padrões de forma a equilibrar as forças envolvidas. É a partir desse conhecimento

que os padrões ajudam os desenvolvedores a melhorarem sua habilidade em modelagem orientada a objetos.

Quanto mais padrões eu utilizar, melhor vai ficar o meu código?

Claro que não! Como o padrão é uma solução para um problema, aplicá-lo onde o problema não existe irá apenas complicar as coisas. Isso seria como para uma pessoa que acabou a aprender a usar um martelo, ver todos os problemas como se fossem um prego. Por mais que um parafuso possa parecer com um prego, a solução é bem diferente.

O mesmo vale para os padrões. Não é só porque você aprendeu um novo padrão, que precisa ir utilizando-o em todas as soluções. Lembre-se que eles também possuem consequências negativas que podem se sobrepor às vantagens em alguns casos. Procure possuir diversos padrões em sua caixa de ferramentas e utilizar o mais adequado para a cada situação. Sua utilização desnecessária pode ser desastrosa e gerar uma sobre-engenharia do sistema, o que, no mínimo, dificulta a manutenção do código.

Os padrões também adicionam à equipe uma terminologia para se referir a soluções de modelagem, criando um vocabulário compartilhado. Dessa forma, quando alguém mencionar o nome de um padrão, todos saberão do que se trata. Esse nome não se refere apenas à estrutura, mas a todas informações agregadas a ele, como sua dinâmica e suas consequências.

Este livro não irá descrevê-los em um formato tradicional, mas irá utilizá-los para exemplificar técnicas que podem ser utilizadas para projetar um software. Cada capítulo irá focar em uma técnica ou em um tipo de problema, e então serão apresentados os padrões relacionados com exemplos que irão ilustrar sua implementação. Também serão discutidas as diferentes alternativas de solução e as consequências positivas e negativas trazidas por cada uma delas. O objetivo é apresentar importantes técnicas de projeto orientado a objetos a partir da aplicação de padrões.

Todos os padrões apresentados no livro utilizarão **`este estilo`**, então toda vez que vir um nome nesse estilo, saiba que está fazendo referência a um padrão. Seus nomes serão mantidos em inglês devido ao fato de serem referenciados dessa forma pela comunidade de desenvolvimento de software no Brasil, sendo que para muitos

deles não existe uma tradução adequada. Os capítulos do livro podem ser lidos de forma independente, porém eles podem fazer referência a padrões explicados em capítulos anteriores. Para desenvolvedores inexperientes, recomenda-se a leitura dos capítulos na ordem de apresentação.

Grande parte dos padrões utilizados nesse livro são os contidos no [10], porém outros importantes de outras fontes também estão incluídos. Quando nenhuma referência for provida com a primeira citação ao padrão, isso significará que é um padrão desse livro.

Chegando aos Padrões pela Refatoração

Existem basicamente duas formas para se implementar os padrões no seu código. A primeira delas, é a sua introdução na modelagem antes do código ser desenvolvido. Nesse caso, é feita uma análise preliminar do problema, em que é identificado o contexto no qual é adequada a aplicação do padrão. A outra abordagem é através da refatoração [12], uma técnica no qual o código é reestruturado de forma a não haver modificação de seu comportamento. Nesse contexto, o código já foi desenvolvido porém apresenta algum tipo de deficiência em termos de projeto, situação conhecida como **mau cheiro**. Sendo assim, o código vai sendo alterado em pequenos passos até que se chegue ao padrão alvo.

Saber como utilizar refatoração para se chegar a padrões é uma técnica importante [16], pois nem sempre a solução mais adequada é a que é empregada na primeira vez. Esse livro irá apresentar, juntamente com os padrões, como pode surgir a necessidade de sua aplicação em um código existente e quais os passos de refatoração que podem ser feitos para sua implementação. No exemplo apresentado nesse capítulo, foi mostrado um código problemático já existente que foi refatorado para implementar o padrão `Strategy`. Os passos da refatoração para ele serão mostrados no Capítulo 3.

1.6 Como o Livro Está Organizado

Esta seção irá fechar esse capítulo inicial, falando um pouco sobre a organização do livro. Aqui os padrões são apresentados de uma forma distinta de outros livros. Os capítulos são organizados de acordo com o princípio de design utilizado pelo padrão. Dessa forma, eles servirão para exemplificar tipos de problema que podem ser resolvidos com aquela técnica. Serão apresentados exemplos, que além de mostrar a aplicação do padrão de forma prática também irão ilustrar como eles podem ser

combinados.

O Capítulo 2 começa explorando de forma mais profunda a utilização de herança e quais são os padrões que fazem uso desse princípio para permitir a extensão de comportamento. De forma complementar, o Capítulo 3 contrasta o uso da herança com o uso da composição, ressaltando as principais diferenças entre elas. Nesse ponto, diversos padrões que fazem uso da composição serão apresentados, inclusive de forma combinada com a herança.

Seguindo a linha de raciocínio, o Capítulo 4 apresenta a composição recursiva, e como ela pode ser utilizada para facilitar o reaproveitamento de código. A partir dos padrões que utilizam esse princípio é possível combinar o comportamento de diversas classes de formas diferentes, obtendo uma grande quantidade de possibilidades de comportamento final. Em seguida, o Capítulo 5 mostra como a composição recursiva também pode ser utilizada para o encapsulamento de objetos, de forma a permitir que funcionalidades sejam adicionadas na classe de forma transparente aos seus clientes.

O Capítulo 6 trata da criação de objetos e dos diferentes padrões que podem ser utilizados para resolver problemas, como uma lógica de criação complexa ou a criação de objetos de classes relacionadas. Ainda tratando desse tema, o Capítulo 7 aborda questões de modularidade dentro da aplicação, apresentando diferentes técnicas para permitir que uma classe obtenha a instância de outra sem depender diretamente dela. Isso permite que novas implementações possam ser plugadas sem a necessidade de recompilar a aplicação.

Em seguida, enquanto os capítulos iniciais apresentaram técnicas para alterar a implementação e, consequentemente, o comportamento de funcionalidades existentes, o Capítulo 8 mostra técnicas para que novas funcionalidades e operações possam ser adicionadas em classes existentes. E, de forma complementar, o Capítulo 9 mostra como a complexidade de uma grande aplicação pode ser gerenciada através da criação de subsistemas e do encapsulamento da gerência da comunicação entre os objetos.

Finalizando o livro, o Capítulo 10 não apresenta nenhum novo padrão, mas alguns tópicos avançados e complementares ao que foi apresentado ao longo de todo livro. Esses tópicos incluem questões sobre o desenvolvimento de frameworks, utilização de tipos genéricos em padrões e da utilização de padrões em um processo TDD. Por fim, ele termina falando um pouco da cultura da comunidade que existe no Brasil e no mundo para o estudo e identificação de novos padrões.

Capítulo 2

Reuso Através de Herança

"Entidades de software devem ser abertas a extensão, mas fechadas a modificação."
– Bertrand Meyer

A herança é uma das principais funcionalidades de linguagens orientadas a objetos. É a partir dela que é possível grande parte do potencial de reúso. O problema é que muita gente que diz isso para por aí ainda busca o reúso através da herança de forma errada.

O primeiro pensamento que normalmente temos ao se estender uma classe é reutilizar a lógica da superclasse na subclasse. Será que isso é mesmo verdade? O que uma subclasse pode reutilizar da superclasse? A estrutura de dados? Seus métodos? A reutilização da estrutura de dados na herança não é muito importante, pois simplesmente instanciando e utilizando uma classe isso é possível de ser feito. A utilização dos métodos da superclasse pela subclasse é equivalente ao reúso de funções na programação estruturada, não sendo também grande novidade.

O potencial de reúso possível com herança está em outro local! Ele pode estar sim no reúso de código da superclasse, porém não é com a subclasse chamando mé-

todos da superclasse, mas com a superclasse chamando código da subclasse. Quando um método da superclasse chama um método que pode ou deve ser implementado na subclasse, isso permite que um mesmo algoritmo possa ser reutilizado com passos alterados. Essa flexibilidade aumenta o potencial de reutilização pois permite a sua adaptação para necessidades mais específicas.

Outro local onde o código pode ser reusado é nas classes que utilizam uma variável com o tipo da superclasse. Nesse caso, como as instâncias das subclasses podem ser atribuídas a essa variável, é possível adaptar o comportamento segundo a instância utilizada. Resumindo em apenas uma palavra: polimorfismo! Por exemplo, o algoritmo de ordenação de listas pode ser reutilizado em diversas aplicações para ordenação de listas de diversas classes. Isso só é possível pois todas as classes compartilham a mesma abstração do tipo Comparable.

O objetivo desse capítulo é mostrar como a herança pode ser utilizada em um design orientado a objetos para permitir adaptação de comportamento e consequentemente um maior reúso. Isso será feito apresentando padrões que utilizam os princípios da herança descritos e como eles podem ser utilizados para criação de soluções.

2.1 Exemplo de Padrão que Utiliza Herança - Null Object

"O que é invisível para nós é também crucial para nosso bem estar."
– Jeanette Winterson

Para começar a falar de herança, essa seção apresenta um padrão bem simples, porém bastante útil. Apesar de não ter sido incluído no GoF, em uma entrevista recente, um dos autores confessou que o colocaria em uma hipotética segunda edição do livro [20]. Ele irá ilustrar como com o uso da herança é possível *"enganar"* o código que utiliza a classe, introduzindo um novo comportamento que irá eliminar a necessidade do uso de condicionais. Como para o código cliente o comportamento da classe é invisível, o dinamismo desse comportamento pode ser a chave para se lidar com situações diferentes.

Vou começar citando uma situação que certamente muitos já viram em algum código que trabalharam. A classe `ApresentacaoCarrinho` apresentada na listagem a seguir chama o método `criarCarrinho()` da classe `CookieFactory` para recuperar um carrinho previamente criado pelo usuário a partir dos cookies armazenados em seu navegador. Uma vez tendo os valores recuperados, são

definidos atributos para a exibição das informações do carrinho na parte superior da página do usuário. O grande problema desse código é que, caso nenhum carrinho tenha sido criado, ele retorna um valor nulo. Daí é preciso adicionar condicionais para configurar os valores adequados para quando o carrinho for nulo.

Listagem 2.1 - Condicionais para Proteção de Valores Nulos:

```
public class ApresentacaoCarrinho{

    public void colocarInformacoesCarrinho(HTTPServletRequest request) {
        Carrinho c = CookieFactory.criarCarrinho(request);
        if(c != null) {
            request.setAttribute("valor", c.getValor());
            request.setAttribute("qtd", c.getTamanho());
        } else {
            request.setAttribute("valor", 0.0);
            request.setAttribute("qtd", 0);
        }
        if(request.getAttribute("username") == null) {
            if (c != null) {
                request.setAttribute("userCarrinho",
                                     c.getNomeUsuario());
            } else {
                request.setAttribute("userCarrinho",
                                     "<a href=login.jsp>Login</a>");
            }
        } else {
            request.setAttribute("userCarrinho",
                                 request.getAttribute("username"));
        }
    }
}
```

Se esse problema se tornar recorrente, ele pode ser uma verdadeira catástrofe na legibilidade do código de uma aplicação. É comum ver condicionais repetidos para tratar em vários pontos a possibilidade de uma determinada variável ser nula. Esses condicionais garantem a segurança do código apenas naquele ponto, pois nada garante que esse valor será tratado adequadamente em outros locais. Mas como então fazer para proteger minha aplicação de um `NullPointerException` sem precisar recorrer a esse tipo de condicional?

Criando uma Classe para Representar Valores Nulos

O padrão **`Null Object`** [27] propõe a criação de uma classe para representar objetos nulos em uma aplicação. Essa classe deve estender a classe original e implementar seus métodos de forma a executar o comportamento esperado da aplicação quando um valor nulo for recebido. Dessa forma, em vez de se retornar um valor nulo, retorna-se uma instância dessa nova classe.

A classe `CarrinhoNulo` apresentada na listagem a seguir exemplifica a criação de um **`Null Object`** para esse contextoo. Observe que os métodos retornam exatamente os valores que eram configurados para o caso do carrinho ser nulo. Nesse exemplo, apenas valores são retornados, porém em outros casos é preciso ter uma lógica a ser executada dentro dos métodos.

Listagem 2.2 - Definição da Classe para Representar o Carrinho Nulo:

```
public class CarrinhoNulo extends Carrinho{
    public double getValor(){
        return 0.0;
    }

    public int getTamanho(){
        return 0;
    }

    public String getNomeUsuario(){
        return "<a href=login.jsp>Login</a>";
    }
}
```

Na listagem a seguir, é possível observar como o código fica mais simples com a eliminação da parte responsável pelo tratamento dos valores nulos. Outra coisa que precisaria ser alterada é a própria classe `CookieFactory` que deve retornar uma instância de `CarrinhoNulo` em vez de `null` quando nenhum carrinho do usuário for encontrado.

Listagem 2.3 - Código após a introdução do Null Object:

```
public class ApresentacaoCarrinho{
    public void colocarInformacoesCarrinho(HTTPServletRequest request) {
        Carrinho c = CookieFactory.criarCarrinho(request);
```

```
        request.setAttribute("valor", c.getValor());
        request.setAttribute("qtd", c.getTamanho());

        if(request.getAttribute("username") == null) {
            request.setAttribute("userCarrinho", c.getNomeUsuario());
        } else {
            request.setAttribute("userCarrinho",
                                    request.getAttribute("username"));
        }
    }
}
```

Uma consequência interessante da aplicação desse padrão é que ele resolve o problema do tratamento de valores nulos em qualquer ponto da aplicação que utilize essa classe. Por mais que o método apresentado no exemplo tratasse esses valores, a mesma situação poderia acontecer em partes do código que não o fazem, gerando a possibilidade de erro. Apesar das vantagens, o tratamento dos valores nulos não ficam explícitos e isso pode gerar uma certa confusão na hora de ler e dar manutenção nesse código.

O Uso da Herança no Null Object

Uma das coisas interessantes desse exemplo é como o código que utilizava o **`Null Object`** desconhecia completamente que ele estava ali. Ele conhecia apenas a interface da superclasse, mas utilizou as funcionalidades da subclasse. Na verdade, esse é o principal motivo para a utilização da herança no projeto de um software: o uso do polimorfismo. Com ele é possível reutilizar o código cliente para diversas implementações. Inclusive, um bom teste para verificar se o uso de herança é adequado em uma determinada situação é ver se faz sentido substituir a implementação por uma de suas subclasses em todos os contextos.

Herdar ou não herdar de JPanel? Eis a questão!

Eu vejo essa questão da utilização ou não da herança é no desenvolvimento de aplicações desktop em Swing. Ao se criar uma nova tela para aplicação, uma dúvida comum é se essa classe deve estender a classe `JPanel` ou simplesmente utilizá-la. A extensão de `JPanel` pode fazer sentido a princípio, visto que a nova classe será um painel (obedecendo a famosa regra *"é um"*) e que pode ser adicionada em uma interface gráfica.

Por outro lado, se pensarmos nos princípios da herança, uma subclasse deve poder ser utilizada no lugar de sua superclasse. Raciocinando dessa forma, é muito fácil pensar em diversas situações em que a classe com a tela de uma aplicação não faria sentido de ser utilizada no lugar de um painel genérico. Isso também quebraria o contrato da superclasse, visto que diversos métodos, como para adição de componentes e configuração de layout deixariam de fazer sentido.

Somente verificar se a subclasse *é uma* superclasse pode não ser suficiente. Antes de utilizar herança verifique se o polimorfismo faz sentido, ou seja, se qualquer subclasse pode ser utilizada no lugar da superclasse. Em caso negativo, isso é um indício de que a herança está sendo utilizada de forma inadequada. Esse é conhecido como o **Princípio de Substituição de Liskov**, que defende que se uma classe é um subtipo de outra, então os objetos dessa classe podem ser substituídos pelos objetos do subtipo sem que seja necessário alterar as propriedades do programa.

O **`NullObject`** é um padrão que demonstra bem essa características da herança, pois ele permite que o código cliente possa ser utilizado, mesmo para o caso de um objeto nulo. Observe que outras implementações de `Carrinho`, como um que talvez acessa informações remotamente, também poderiam ser retornadas pela fábrica e utilizadas livremente.

2.2 Hook Methods

Um importante uso que pode ser feito da herança é para permitir a especialização de comportamento. Dessa forma, a superclasse pode fornecer uma base para uma determinada funcionalidade, a qual invoca um método que somente é definido pela

superclasse. Esse método funciona como um ponto de extensão do sistema é chamado de método-gancho, ou em inglês, **hook method**.

A Figura 2.1 representa o conceito de *hook method*. A superclasse possui um método principal público que é invocado pelos seus clientes. Esse método delega parte de sua execução para o *hook method*, que é um método abstrato que deve ser implementado pela subclasse. Ele funciona como um "gancho" no qual uma nova lógica de execução para a classe pode ser "pendurada". Cada subclasse o implementa provendo uma lógica diferente. Como essa lógica pode ser invocada a partir do mesmo método público, definido na superclasse, os *hook methods* permitem que o objeto possua um comportamento diferente de acordo com a subclasse instanciada.

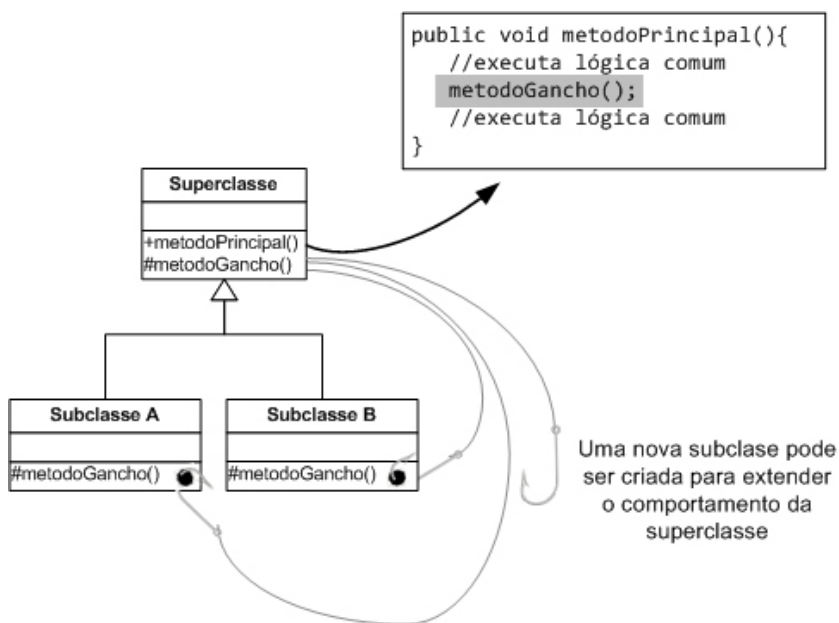



Figura 2.1: Representação de Hook Methods

Essa prática é muito utilizada em frameworks para permitir que as aplicações possam especializar seu comportamento para seus requisitos. Nesse caso, o framework provê algumas classes com *hook methods* que precisam ser especializadas pela aplicação. Sendo assim, a aplicação deve estender essa classe e implementar o *hook method* de forma a inserir o comportamento específico de seu domínio. Imagine, por exemplo, um framework que realiza o agendamento de tarefas. Usando

essa estratégia ele poderia prover uma classe para ser estendida, na qual precisaria ser implementado um *hook method* para a definição da tarefa da aplicação. Enquanto isso, na classe do framework estaria a lógica de agendamento que chamaria o método definido pela aplicação no momento adequado.

Um exemplo de uso dessa técnica está na Servlets API [21], na qual um servlet precisa estender a classe `HTTPServlet`. Essa classe possui o método `service()` que é invocado toda vez que ele precisa tratar uma requisição HTTP. Esse método chama outros métodos, como `doPost()` ou `doGet()`, que precisam ser implementados pela subclasse. Neles ela deve inserir a lógica a ser executada para tratar a requisição recebida. Nesse caso, esses métodos possuem uma implementação *default* vazia e precisam ser implementados somente se necessário.

2.3 Revisando modificadores de métodos

Quando aprendemos o básico da linguagem, aprendemos modificadores de acesso e outros tipos de modificadores de métodos e qual o seu efeito nas classes. O que nem todos compreendem é quando cada um deles precisa ser utilizado e qual o tipo de mensagem que se passa com cada um em termos de design. A seguir estão relacionados alguns modificadores de acesso e qual o recado que uma classe manda para duas subclasses quando utiliza cada um deles:

- `abstract`: Um método abstrato precisa obrigatoriamente ser implementado em uma subclasse concreta. Isso significa que esse é um *hook method* que foi definido na superclasse e obrigatoriamente precisa ser definido.

- `final`: De forma contrária, um método do tipo final não pode ser sobrescrito pelas subclasses e por isso nunca irá representar um *hook method*. Esses métodos representam funcionalidades da classe que precisam ser imutáveis para seu bom funcionamento. Costumam ser os métodos que invocam os *hook methods*.

- `private`: Os métodos privados só podem ser invocados dentro da classe que são definidos. Dessa forma, eles representam métodos de apoio internos da classe e nunca poderão ser substituídos pela subclasse como um *hook method*.

- `protected` e `public`: Todos os métodos públicos e protegidos, desde que não sejam `final`, são candidatos a *hook method*. É isso mesmo! Qualquer

método que pode ser sobrescrito na subclasse pode ser utilizado para a inserção de comportamento. Algumas classes possuem implementações vazias para esses métodos, o que faz com que sejam *hook methods* com implementação opcional pelas subclasses.

Ao criar uma estrutura de classes, é importante colocar os modificadores corretos nos métodos de forma a passar a mensagem certa para as classes que forem estendê-la. Por exemplo, se um método que coordena a chamada de *hook methods* for sobrescrito, eles podem não ser chamados e algumas premissas assumidas na superclasse podem não ser mais verdadeiras. Por isso é preciso muito cuidado ao prover uma classe para ser estendida como parte da API de um componente ou framework.

2.4 Passos diferentes na Mesma Ordem - Template Method

"No meio do caminho tinha uma pedra / tinha uma pedra no meio do caminho / tinha uma pedra / no meio do caminho tinha uma pedra."
– Carlos Drummond de Andrade

O principal padrão que utiliza *hook methods* como técnica é o **`Template Method`**. Este padrão é aplicável quando se deseja definir um algoritmo geral, que estabelece uma série de passos para cumprir um requisito da aplicação. Porém, seus passos podem variar e é desejável que a estrutura da implementação forneça uma forma para que eles sejam facilmente substituídos.

Imagine, por exemplo, como seria se fôssemos definir um algoritmo para as pessoas acordarem e irem ao trabalho. Esse algoritmo envolveria ações como acordar, ir ao banheiro, comer alguma coisa, trocar de roupa e se locomover até o trabalho. Dependendo da pessoa cada um dos passos pode ser diferente, como o tipo de roupa que colocam para ir ao trabalho ou o que comem pela manhã. Na locomoção ao trabalho, alguns podem ir de transporte público, outros podem ir em seu carro e até mesmo a pé ou de bicicleta. Por mais que cada um dos passos seja diferente, o algoritmo que os utiliza acaba sendo o mesmo.

Estrutura do Template Method

Para entender a estrutura do **`Template Method`** vamos utilizar como metáfora algo familiar a muitas pessoas: modelos de documento. Conforme ilustrado na

Figura 2.2, um modelo de documento define algumas partes que são fixas e outras partes que devem ser introduzidas quando o documento propriamente dito for criado. São lacunas que precisam ser completadas e que irão variar de documento para documento. O modelo provê uma estrutura que pode ser facilmente reaproveitada, simplificando sua criação.

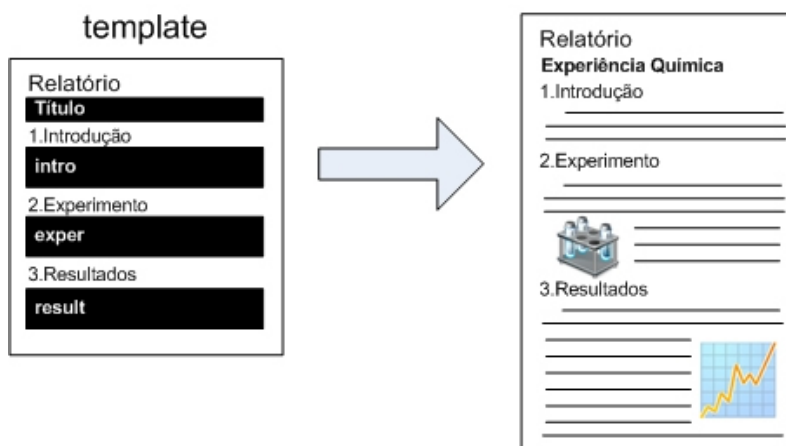



Figura 2.2: Uso de Templates para Documentos

De forma similar, um **`Template Method`** é um modelo de algoritmo que possui algumas partes fixas e algumas partes variáveis. As partes variáveis são lacunas que precisam ser completadas para que o algoritmo faça realmente sentido. As lacunas são representadas como *hook methods* que podem ser implementados nas subclasses. Caso seja uma lacuna obrigatória, o método deve ser definido como abstrato e caso a implementação seja opcional, o método pode ser concreto e normalmente possui uma implementação vazia. O algoritmo é representado através de um método na superclasse que coordena a execução dos *hook methods*.

A Figura 2.3 apresenta a estrutura do padrão **`Template Method`**. A `ClasseAbstrata` representa a superclasse que implementa o **`TemplateMethod`** e que define quais são os *hook methods*. A `ClasseConcreta` representa a classe que herda o **`Template Method`** da `ClasseAbstrata` e define uma implementação concreta dos *hook methods*. A classe representada como `Cliente` invoca o `metodoTemplate()`. Observe que apesar do tipo da variável ser do tipo da classe abstrata, o tipo instanciado é o da subclasse que implementa os passos concretos do algoritmo.

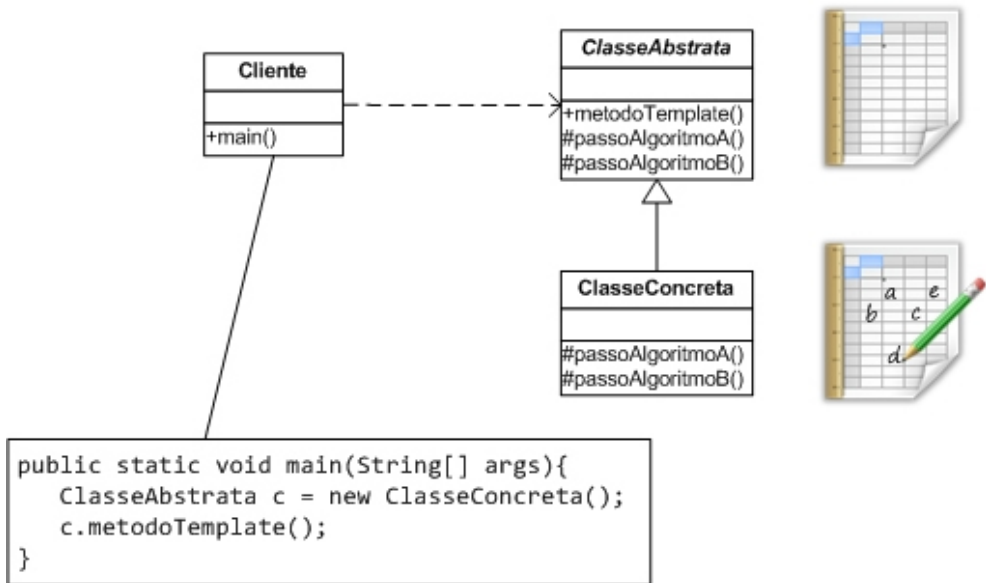



Figura 2.3: Estrutura do Padrão Template Method

Qual a diferença entre Hook Methods e o Template Method?

Muitas pessoas nesse ponto podem estar achando que *Hook Methods* e o padrão **`Template Method`** são a mesma coisa. A grande diferença é que os *Hook Methods* são uma técnica para permitir a extensão de comportamento e o **`Template Method`**, como um padrão, é uma solução para um problema mais específico. Seria correto dizer que o **`Template Method`** utiliza *Hook Methods* em sua solução. O que é importante perceber é que o conceito de *Hook Method* é mais geral e inclusive é utilizado por outros padrões que serão vistos mais à frente nesse livro.

Serializando Propriedades em Arquivos

Imagine que em uma aplicação precisamos pegar mapas de propriedades e serializar em arquivos. A questão é que o formato em que as propriedades precisam

ser estruturadas são diferentes. Por exemplo, pode ser possível estruturar as informações em arquivos XML e em arquivos de propriedades. Além disso, os arquivos podem precisar receber um pós-processamento, como uma criptografia ou uma compactação. Para o exemplo que será apresentado iremos considerar que existem duas possibilidades: a criação do arquivo XML compactado e a criação do arquivo de propriedades criptografado.

Como nesse caso temos um algoritmo base em que os passos podem ser modificados de acordo com a implementação, esse é um cenário adequado para implementação do padrão **Template Method**. A classe `GeradorArquivo`, apresentada na listagem a seguir, possui o método `gerarArquivo()` que define um algoritmo base para a criação do arquivo, invocando os dois métodos abstratos `processar()` e `gerarConteudo()`. Esse método possui o modificador `final` para que ele não possa ser sobrescrito nas subclasses. Note que o método `processar()` fornece uma implementação *default* que retorna o próprio array de bytes, representando o caso em que não existe processamento adicional após o arquivo ser gerado. Diferentemente, o método `gerarConteudo()` é abstrato e obrigatoriamente precisará ser implementado.

Listagem 2.4 - Classe que define o Template Method:

```
public abstract class GeradorArquivo {
    public final void gerarArquivo(String nome,
                                   Map<String, Object> propriedades)
            throws IOException {
        String conteudo = gerarConteudo(propriedades);
        byte[] bytes = conteudo.getBytes();
        bytes = processar(bytes);
        FileOutputStream fileout = new FileOutputStream(nome);
        fileout.write(bytes);
        fileout.close();
    }

    protected byte[] processar(byte[] bytes) throws IOException{
        return bytes;
    }

    protected abstract String
                    gerarConteudo(Map<String, Object> propriedades);
}
```

As duas próximas listagens apresentam as classes `GeradorXMLCompactado` e `GeradorPropriedadesCriptografado` que implementam a classe `GeradorArquivo`, fornecendo uma implementação para os passos do algoritmos definidos na superclasse. A primeira implementação, no método `gerarConteudo()` cria um arquivo XML no qual existe uma tag chamada `<properties>` e cada propriedade é um elemento dentro dessa tag. O método `processar()` usa a classe `ZipOutputStream` para gerar um arquivo compactado. Já na segunda listagem, o método `gerarConteudo()` cria a estrutura de um arquivo de propriedades em que cada linha possui o formato *propriedade=valor*. O método de processamento criptografa os bytes do arquivo utilizando um algoritmo simples chamado cifra de César, em que o valor de cada byte é deslocado de acordo com o parâmetro `delay`.

Listagem 2.5 - Classe que gera o arquivo com XML compactado:

```
public class GeradorXMLCompactado extends GeradorArquivo {
    protected byte[] processar(byte[] bytes) throws IOException {
        ByteArrayOutputStream byteOut = new ByteArrayOutputStream();
        ZipOutputStream out = new ZipOutputStream(byteOut);
        out.putNextEntry(new ZipEntry("internal"));
        out.write(bytes);
        out.closeEntry();
        out.close();
        return byteOut.toByteArray();
    }

    protected String gerarConteudo(Map<String, Object> props) {
        StringBuilder propFileBuilder = new StringBuilder();
        propFileBuilder.append("<properties>");
        for(String prop : props.keySet()){
            propFileBuilder.
                append("<"+prop+">"+props.get(prop)+"</"+prop+">");
        }
        propFileBuilder.append("</properties>");
        return propFileBuilder.toString();
    }
}
```

Listagem 2.6 - Classe que gera arquivo de propriedades criptografado:

```
public class GeradorPropriedadesCriptografado extends GeradorArquivo {
    private int delay;

    public GeradorPropriedadesCriptografado(int delay) {
        this.delay = delay;
    }

    protected byte[] processar(byte[] bytes) throws IOException {
        byte[] newBytes = new byte[bytes.length];
        for(int i=0;i<bytes.length;i++){
            newBytes[i]= (byte) ((bytes[i]+delay) % Byte.MAX_VALUE);
        }
        return newBytes;
    }

    protected String gerarConteudo(Map<String, Object> props) {
        StringBuilder propFileBuilder = new StringBuilder();
        for(String prop : props.keySet()){
            propFileBuilder.append(prop+"="+props.get(prop)+"\n");
        }
        return propFileBuilder.toString();
    }
}
```

Consequências do Uso do Template Method

Pelo exemplo apresentado e pela descrição do padrão, é possível perceber que com o **Template Method** podemos reaproveitar o código relativo à parte comum de um algoritmo, permitindo que cada passo variável possa ser definido na subclasse. Isso também é uma forma de permitir que a funcionalidade da classe que define o algoritmo básico seja estendida. Assim é possível definir uma funcionalidade mais geral na qual pode ser facilmente incorporada a parte específica do domínio da aplicação. É importante lembrar que os modificadores adequados dos métodos devem ser utilizados para impedir que o contrato da superclasse com os seus clientes seja quebrado.

Porém, como foi dito no primeiro capítulo, o uso da herança nesse padrão também traz algumas limitações. A primeira é que a herança *"é uma carta que só pode ser jogada uma vez"*, isso significa que uma classe que precise de comportamentos de duas outras classes só poderá fazer o uso da herança para uma delas. Essa questão

será melhor discutida no Capítulo 3. Outra questão é que depois que uma implementação for instanciada não será mais possível alterar os passos do algoritmo.

Ao utilizar um padrão, é preciso avaliar para os requisitos de sua aplicação quais consequências pesam mais ou menos. A partir dessas informações é possível decidir se seu uso será ou não adequado. Ressalto que o maior problema de uma solução que possui limitações é quando elas são desconhecidas pelo desenvolvedores, pois quando se tem consciência de sua existência é possível gerenciar o risco ou tratá-las, muitas vezes a partir de outros padrões.

2.5 Refatorando na Direção da Herança

Muitas vezes, os requisitos que direcionam o desenvolvimento para o uso da herança não surgem todos ao mesmo tempo. Outras vezes, simplesmente não se visualiza no momento que se está programando que aquele padrão pode ajudar na solução. Nesses casos, o padrão pode ser alcançado a partir de refatorações. A ideia é conseguir, a partir de pequenos passos, ir modificando aos poucos a estrutura da solução, sem modificar o seu comportamento. Vale ressaltar que é extremamente recomendado que a refatoração seja apoiada por uma suíte de testes automatizados, que devem ser executados a cada passo da refatoração para verificar se o comportamento foi mantido.

Demolindo tudo que foi criado e reconstruindo do zero

Um grande erro ao se refatorar um código é destruir toda a solução existente e reconstruir do zero uma nova solução. Isso é ruim pois durante um longo periodo de tempo perde-se a referência dos testes automatizados, que dão o feedback se o comportamento está sendo mantido inalterado. No decorrer desse livro, serão apresentados os passos que devem ser seguidos para se refatorar a solução na direção dos padrões apresentados. Com isso, pretende-se mostrar como que com uma sequência de pequenas mudanças pode-se alcançar um grande impacto na qualidade do código e na modelagem da aplicação.

No caso de necessidade de se refatorar para o uso da herança, existem dois cenários diferentes. No primeiro, toda a funcionalidade foi acumulada em apenas uma

classe, que possui um grande número de condicionais e um código de difícil manutenção. O segundo cenário ocorre quando funcionalidades com similaridades nos algoritmos são implementadas em diferentes classes, gerando duplicação de código.

A Figura 2.4 apresenta os passos para a refatoração quando uma única classe concentra em um método todas as possibilidades do algoritmo utilizando condicionais. A primeira etapa é extrair os métodos de acordo com os passos que devem ser realizados pelo algoritmo. Após a extração, cada método ainda possuirá a lógica condicional necessária para todos os possíveis caminhos. Na etapa seguinte é criada uma subclasse para representar uma alternativa de execução do algoritmo e os passos seguintes consistem em mover a funcionalidade para essa subclasse. Esse processo vai sendo repetido até que cada possível caminho seja movido para uma subclasse e somente a lógica mais geral esteja concentrada na superclasse.

Para que isso seja feito, o primeiro passo é sobrescrever os métodos extraídos na nova subclasse criada. A princípio, esses métodos ainda irão delegar a execução da sua funcionalidade para a superclasse, fazendo uma chamada ao método original na forma `super.orginal()`. Caso exista algum parâmetro setado na superclasse para a escolha dos passos do algoritmo, ele pode ser configurado de forma fixa na subclasse, de acordo com o que será implementado nela. A etapa seguinte consiste em mover para a subclasse a lógica relativa aos métodos com os passos do algoritmo. Deve ser retirado o condicional na superclasse, juntamente com o código a ser executado, e inserido no método da subclasse, que não deve mais delegar sua execução para superclasse. Esse procedimento deve ser feito método por método. Em seguida, os mesmos passos devem ser repetidos para a criação de novas subclasses, até que não exista mais lógica nesses métodos da superclasse. No caso de não haver uma implementação *default*, eles devem se tornar abstratos.

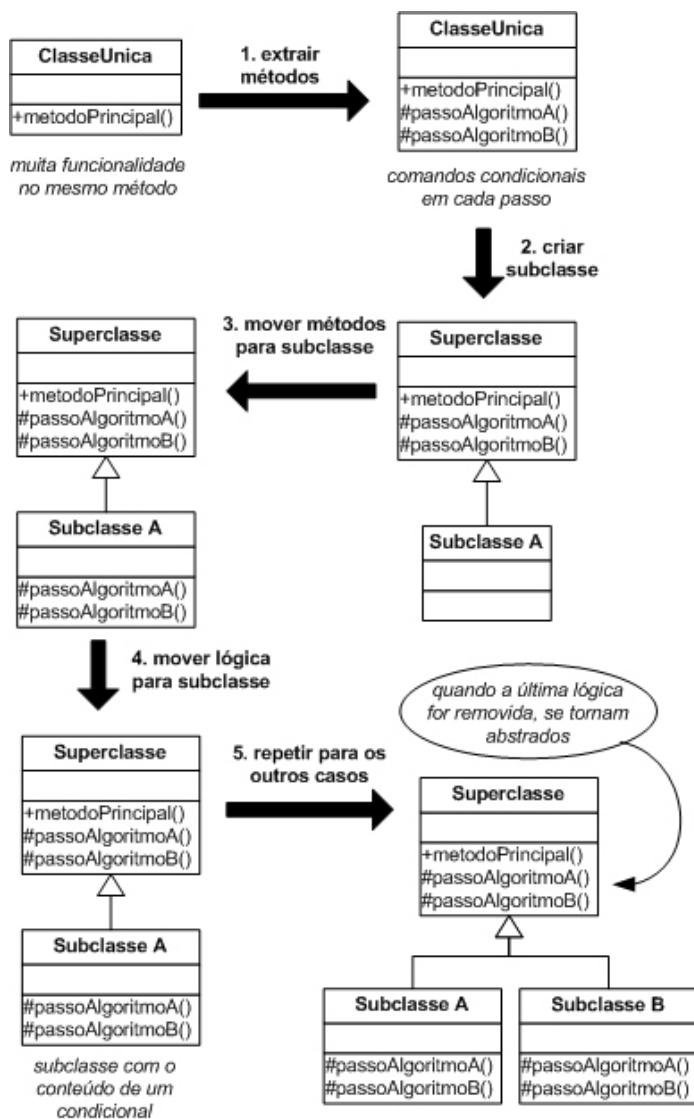



Figura 2.4: Quebrando classe única com método grande e complexo

A outra possibilidade de refatoração deve ser realizada quando existirem duas classes que implementam algoritmos similares e possuam código duplicado. Conforme está apresentado na Figura 2.5, a refatoração irá levar a refatoração na direção da definição de uma superclasse comum que implementa um `Template Method`. A primeira coisa que precisa ser feita é a extração de métodos com as partes que dife-

renciam um algoritmo do outro. Depois dessa etapa, o código interno dos métodos principais deve ser igual e todas as diferenças devem estar nos métodos extraídos. Em seguida, para uniformizar as classes, é preciso renomear os métodos para que fiquem com mesmo nome e mesma assinatura em ambas as classes (incluindo tipos de parâmetros e exceções). Deve-se buscar nomes que representem bem a parte do algoritmo abstraída por eles.

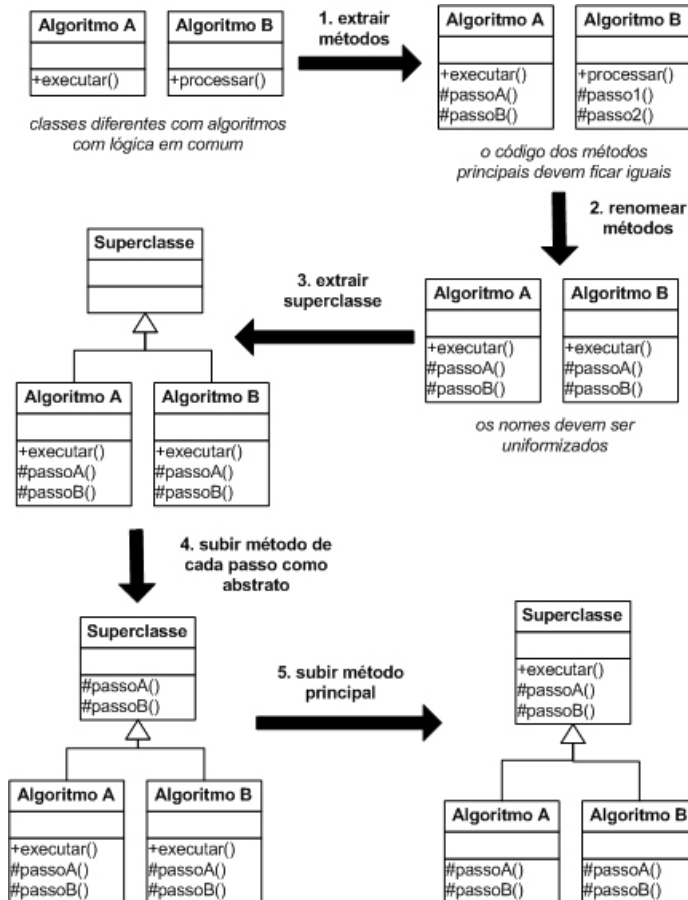



Figura 2.5: Criando superclasse para duas classes com código duplicado

Para seguir em direção a implementação do **Template Method**, o passo seguinte é criar uma superclasse que seja especializada por ambas as classes. A princípio essa classe estará vazia para que não conflite com a implementação de cada uma. Em

seguida, deve-se subir para superclasse os métodos extraídos, porém como métodos abstratos. Isso permitirá que na etapa seguinte o método principal seja totalmente movido para superclasse e retirado de todas as subclasses.

Mau cheiro de código: Missing Template Method

Na terminologia de design de software, um **mau cheiro**, ou *bad smell*, se refere a algum indício de problemas no código que podem significar deficiências em sua modelagem. Um mau cheiro é diferente de um bug, pois não necessariamente ele causa um erro na aplicação, mas traz outras consequências negativas como dificuldade de manutenção e falta de flexibilidade. Existem ferramentas que fazem a detecção automática de maus cheiros, usando como base suas métricas e características. O livro "Refatoração: aperfeiçoando o projeto de código existente" [12] traz uma lista com alguns deles.

Um dos mau cheiros que são frequentemente detectados por essas ferramentas se chama *"Missing Template Method"*. Ele se refere a dois componentes que possuem um número significante de similaridades, mas não compartilham de uma mesma abstração, como uma interface, ou de uma mesma implementação, como um **`Template Method`**. Ao se deparar com essa situação, seja detectada por uma ferramenta ou não, é indicada a refatoração descrita.

2.6 Criando Objetos na Subclasse - Factory Method

"Pequenas surpresas após cada esquina, mas nada perigoso."
– Willy Wonka, A Fantástica Fábrica de Chocolates

O conceito de *hook method* não se aplica somente ao padrão **Template Method**. Existem outros padrões que fazem uso do mesmo princípio para solucionar problemas diferentes. Eles ainda podem combinar isso com outras técnicas para compor soluções mais elaboradas. Essa seção apresenta o padrão **Factory Method**, que é utilizado para resolver um problema relacionado a criação de objetos.

Quando estamos desenvolvendo uma aplicação, é comum termos classes de diferentes hierarquias se relacionando. Por exemplo, uma classe que representa uma entidade do sistema pode se relacionar com a classe que faz a sua validação, ou uma classe que representa um documento pode se relacionar com a classe que gera sua representação em PDF. Quando isso ocorre, acabamos tendo que criar uma classe em função da outra que está sendo utilizada. No exemplo citado, sabemos que todo

documento possui seu gerador de PDF, mas como relacionar as duas classes? Para a classe que estará trabalhando com a lógica de geração de arquivos, não interessa de qual classe é a instância que ele está recuperando, mas apenas que essa instância seja relacionada com a classe que está trabalhando.

Relacionando DAOs e Classes de Serviço

Para ir mais fundo nesse problema da criação de objetos, vamos considerar um exemplo bastante comum em todo tipo de aplicação. Quando temos uma arquitetura de camadas definida, é comum que objetos de uma camada precisem criar os objetos respectivos da camada seguinte. Nesse exemplo, consideraremos duas camadas da aplicação: a camada DAO (nome que vem do padrão **`Data Access Object`**) que realiza o acesso ao banco de dados e a camada de serviço, onde estão implementadas as regras de negócio. Nesse caso, a camada de serviço relacionada com uma entidade deve criar o DAO para a mesma entidade.

A interface `DAO` define os métodos gerais de acesso a dados que serão disponibilizados, como para recuperação, gravação e exclusão de entidades. Essa interface precisará ser implementada para cada entidade do sistema, para a criação dessas operações de cada uma. Note que a interface possui um parâmetro genérico que é utilizado para determinar a entidade associada ao DAO e para que essa entidade seja utilizada nos parâmetros e retornos dos métodos, variando de acordo com que que for setado na implementação.

Listagem 2.7 - Interface de Definição de um DAO:

```
public interface DAO<E> {
    public E recuperarPorId(Object id);
    public void salvar(E entidade);
    public void excluir(Object id);
    public List<E> listarTodos();
}
```

A classe `ServicoAbstrato` representa uma classe da camada de negócios que contém serviços relacionados a uma entidade. Para prover sua funcionalidade, ela precisa da colaboração do DAO relacionado a mesma entidade. Essa classe possui métodos com serviços gerais que são providos a todas as entidades. Um exemplo é o método `gravarEntidadeEmArquivo()`, que recupera uma entidade a partir de seu identificador e chama a instância de GeradorArquivo (aquela classe criada

anteriormente nesse capítulo) para criação de um arquivo com as propriedades da classe. Essas propriedades são obtidas através do método `getMapa()` definido para o tipo Entidade. Vale ressaltar que qualquer subclasse de GeradorArquivo pode ser utilizada, pois ela é passada no construtor da classe.

A grande questão é que os métodos gerais dessa classe abstrata precisam acessar o DAO para executar sua funcionalidade, porém qual será a instância é algo que só será definido na subclasse, quando a entidade com a qual se trabalha já terá sido definida. Dessa forma, a classe definiu um método abstrato chamado `getDAO()`, que retorna a instância do DAO para ser utilizada. Assim, as subclasses devem implementar esse método de forma a retornar a instância correta. Esse é um *hook method* com o objetivo de criar um objeto.

Listagem 2.8 - Serviço abstrato que define um método para recuperação do DAO:

```
public abstract class ServicoAbstrato<E>{
    public GeradorArquivo gerador;

    public ServicoAbstrato(GeradorArquivo gerador){
        this.gerador = gerador;
    }

    public abstract DAO<E> getDAO();

    //Serviço geral
    public void gravarEntidadeEmArquivo(Object id, String nomeArquivo){
        E entidade = getDAO().recuperarPorId(id);
        gerador.gerarArquivo(nomeArquivo,
                            ((Entidade)entidade).getMapa());
    }
}
```

A classe `ServicoProduto`, apresentada na próxima listagem, representa um exemplo de uma subclasse da classe `ServicoAbstrato` para a entidade `Produto`. Observe que ela implementa o *hook method* definido e cria a instância do DAO relacionado com a classe `Produto`. No exemplo, a instância é criada quando o primeiro acesso é feito ao método. Essa classe pode possuir métodos específicos de produto, porém os métodos herdados da superclasse que utilizam o método `getDAO()` também poderão ser utilizados.

Listagem 2.9 - Serviço concreto que define a implementação Factory Method:

```
public class ServicoProduto extends ServicoAbstrato<Produto>{
    private DAO<Produto> dao;

    public DAO<Produto> getDAO(){
        if(dao == null){
            dao = new ProdutoDAO();
        }
        return dao;
    }

    //Funcionalidade específica
    public double getPrecoProduto(Object produtoId){
        Produto p = getDAO().recuperarPorId(id);
        return p.getPreco();
    }
}
```

Vale observar a utilização dos tipos genéricos nesse exemplo. Como a classe `ServicoProduto` especificou de forma explícita o tipo genérico de sua superclasse, todos os elementos da classe que usavam o parâmetro do tipo genérico serão fixados. Como o método `getDAO()` precisará agora retornar um `DAO<Produto>`, haverá um erro de compilação caso um DAO com outro tipo genérico for retornado.

Entendendo o Factory Method

O exemplo de criação do DAO nas classes de serviço, ilustram bem o cenário onde a classe da instância utilizada depende da subclasse. O padrão **Factory Method** utiliza um *hook method* para delegar a criação da instância para a subclasse. Isso permite que métodos mais gerais na superclasse possam utilizar essa instância mesmo sem conhecer a classe concreta. Isso pode ser feito invocando o método abstrato de criação que é implementado na subclasse.

A Figura 2.6 apresenta um diagrama com a estrutura do padrão **Factory Method**. Nesse padrão, a classe principal possui uma dependência com uma abstração de uma hierarquia de classes. No diagrama, `Dependencia` está representada como uma interface, porém não há nada que impede de ser uma classe. O método `metodoFabrica()` possui a responsabilidade de criar uma instância de `Dependencia`, porém na classe principal ele é um método abstrato. Dessa forma,

nas subclasses esse método precisa ser implementado e de deve ser criado um objeto de uma classe concreta que possua a abstração da dependência.

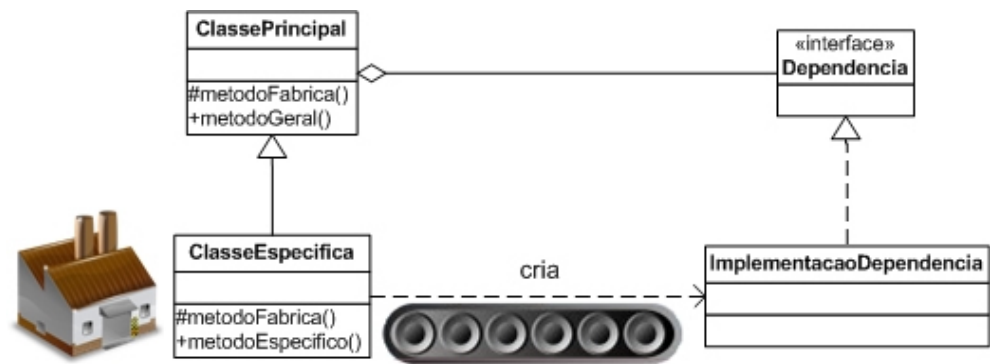



Figura 2.6: Estrutura do padrão Factory Method

A partir desse padrão, é possível desacoplar a superclasse da criação de uma dependência. Com a criação das instâncias na subclasse, apenas elas ficam acopladas as classes concretas da hierarquia. Dessa forma, caso uma nova instância da dependência precise ser utilizada pela superclasse, basta criar uma nova subclasse que retorne aquela instância.

2.7 Considerações Finais do Capítulo

Esse capítulo abordou a utilização da herança para a reutilização de código. O primeiro padrão apresentado foi **`Null Object`**, que mostra como o polimorfismo pode ser utilizado para que o código dos clientes de uma classe possa ser adaptados com suas extensões. Nesse padrão foi visto como a extensão de uma classe pode ser sua representação nula, fazendo com que o código cliente consiga lidar com ela de forma transparente.

Em seguida foi mostrado como o uso da herança pode ser feito para permitir a especialização do código da superclasse. A partir de *hook methods* é possível que as subclasses insiram comportamento em métodos que estão implementados na superclasse. Foram apresentados os padrões **`Template Method`** e **`Factory Method`** que fazem uso desse princípio.

Apesar desses padrões possuírem o uso da herança como parte principal de sua solução, o uso dos princípios apresentados não são sua exclusividade. Em qualquer

outro contexto em que os requisitos puderem ser solucionados a partir da especialização da funcionalidade da superclasse, os mesmo princípios apresentados podem ser utilizados. O próximo capítulo irá explorar o uso da composição e mostrar como ela pode ser utilizada em conjunto com a herança e quando seu uso é mais adequado.

Capítulo 3

Delegando Comportamento com Composição

"A primeira regra do gerenciamento é a delegação. Não tente fazer tudo sozinho porque você não consegue."
– Anthea Turner

Ao pensar em objetos, uma coisa muito natural é imaginar em como são combinados. Na verdade, não existem grandes construções do homem que não são feitas a partir da combinação de diversos objetos. Uma casa, por exemplo, é feita a partir da combinação de diversos blocos. Olhando outros exemplos, um carro é composto de peças e um computador é feito de componentes eletrônicos. O que é mais interessante é que a medida que vamos descendo, muitas vezes o padrão vai se repetindo: uma peça pode ser composta por outras. Uma placa de computador é feita com capacitores, resistências e chips. É a quantidade infinita de combinações desses elementos que torna possível a construção de diversas coisas diferentes a partir do mesmo material.

Quando a programação orientada a objetos foi criada, um dos objetivos era simplificar a abstração dos conceitos do mundo real para o software. Dessa forma, um elemento existente no domínio da aplicação poderia ser representado utilizando um objeto no software. Da mesma forma que as coisas interagem no mundo real para a realização de um objetivo, dentro de um software não poderia ser diferente. Do mesmo jeito que diversas peças são combinadas em vários níveis para compor um componente físico, os componentes de software também podem ser resultado da combinação de outros componentes mais granulares.

Apesar de não estar entre os quatro elementos principais de uma linguagem orientada a objetos, a composição é um conceito que vem implícito quando se fala em objetos. O Capítulo 1 já apresentou o uso da composição através do padrão **`Strategy`**, e este capítulo irá explorar o conceito mais a fundo e mostrar como ele pode ser utilizado para a extensão de comportamento. Será mostrado como a composição combinada com o uso de abstrações pode ser eficaz na solução de diferentes problemas. A primeira seção começa revisitando o exemplo do gerador de arquivos do capítulo anterior para ilustrar cenários onde as limitações da herança são evidentes.

3.1 Tentando Combinar Opções do Gerador de Arquivos

No exemplo do capítulo anterior, a classe `GeradorArquivo` implementava o padrão **`Template Method`** para permitir que as subclasses pudessem especializar parte do comportamento, como o formato da geração do conteúdo e um pós-processamento a ser feito nos bytes do arquivo. O método principal `gerarArquivo()` invocava os *hook methods* `processar()` e `gerarConteudo()`, que eram implementados somente na subclasse. As duas subclasses de `GeradorArquivo`, `GeradorXMLCompactado` e `GeradorPropriedadesCriptografado`, respectivamente implementam os métodos para gerar em XML compactados e de propriedade criptografados.

O problema começa ao tentarmos ter uma implementação que combina o comportamento dessas subclasses, como para a geração de arquivos XML **e** criptografados. Se tentarmos utilizar a herança, acabamos com a duplicação de alguma parte do código. A Figura 3.1 mostra o que acontece quando criamos uma nova classe que define o gerador de XML como uma superclasse com duas subclasses que especializam apenas a parte do pós-processamento. Apesar de ter sido criada uma abstração refe-

rente ao formato do arquivo, a ausência de uma abstração para o pós-processamento faz com que haja duplicação de código.

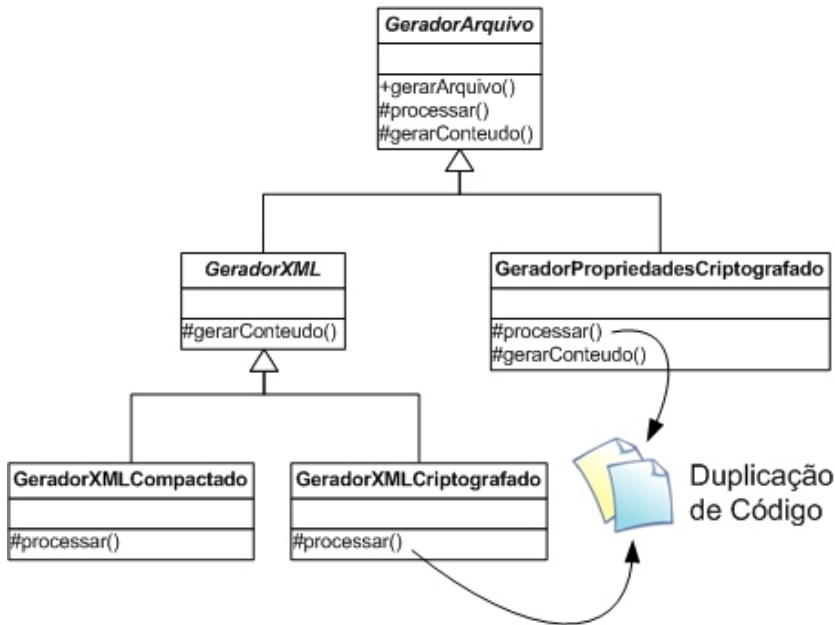



Figura 3.1: Duplicação de código ao se tentar criar um novo nível na hierarquia

A Figura 3.2 mostra que se tentarmos estruturar a hierarquia de uma outra forma para eliminar a duplicação anterior, continuaremos com duplicação em uma outra parte do código.

A limitação do uso da herança, que fica evidente nesse problema, é o fato de ela poder ser utilizada apenas uma vez. Se fosse possível ter herança múltipla em Java, uma classe poderia obter a implementação do formato do arquivo de uma superclasse e a do pós-processamento de outra. Como isso não é possível, quando precisamos de uma implementação de outro ramo da hierarquia, acaba havendo uma duplicação de código.

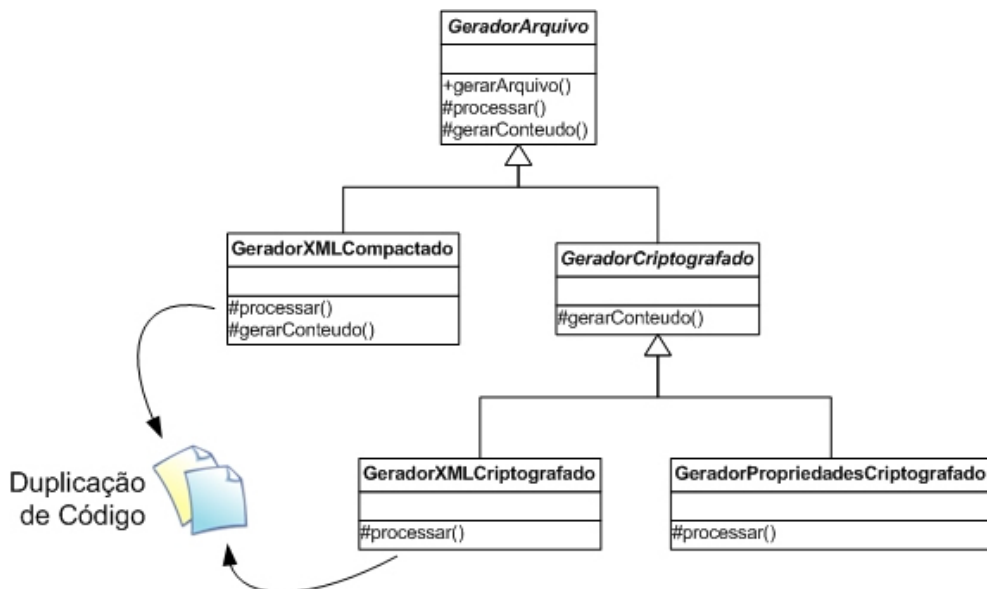



Figura 3.2: Manutenção da duplicação ao se tentar criar a hierarquia pelo outro lado

Quando mais de um *hook method* são definidos em uma superclasse e implementados por uma subclasse, essas implementações ficam ligadas uma a outra devido ao fato de terem sido implementadas na mesma classe. Por mais que seja possível ir implementando cada *hook method* em um nível diferente da hierarquia, a duplicação ainda pode acontecer, principalmente se esses fatores que podem variar - chamados de **variabilidade** - são independentes um do outro. O problema vai se tornando ainda mais crítico quando se aumenta o número de variabilidades e o número de possíveis implementações de cada uma delas. A quantidade de classes necessárias para implementar todas as possibilidades tende a aumentar de forma exponencial!

3.2 Bridge - Uma Ponte entre Duas Variabilidades

"A ponte não é de concreto, não é de ferro / Não é de cimento / A ponte é até onde vai o meu pensamento / A ponte não é para ir nem pra voltar / A ponte é somente pra atravessar / Caminhar sobre as águas desse momento"
– Lenini

Como todos já devem desconfiar pelo foco do capítulo, a composição será utilizada como forma de resolver o problema descrito. Como com a pura utilização de

herança, a implementação de uma variabilidade fica ligada a outra, a ideia da nova solução seria separar a implementação de uma delas em uma outra classe. Da mesma forma mostrada no padrão **Strategy** no Capítulo 1, essa classe irá compor a classe principal, no caso `GeradorArquivo`. Dessa forma, o método implementado nessa hierarquia de classes à parte será invocado como parte da implementação do algoritmo. A Figura 3.3 mostra como ficaria a estrutura de classes com a utilização da composição para os algoritmos de pós-processamento dos arquivos.

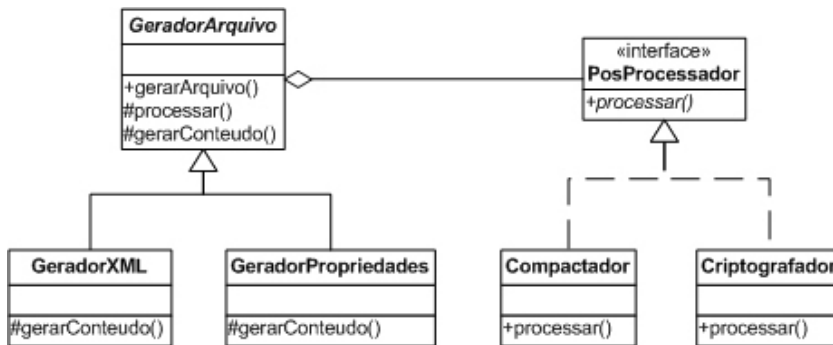



Figura 3.3: Utilização de Composição no GeradorArquivo

A listagem a seguir apresenta o código do `GeradorArquivo` após a aplicação dessa solução. O método `processar()` não está mais na mesma classe e sim na instância de `PosProcessador` que compõe `GeradorArquivo`. Dessa forma, qualquer implementação dessa interface pode ser utilizada para realizar o pós-processamento do arquivo. A grande motivação dessa solução no cenário apresentado é que o processador pode ser configurado independente da subclasse que está sendo utilizada, permitindo às duas variar de forma independente.

Listagem 3.1 - Classe GeradorArquivo utilizando composição:

```
public abstract class GeradorArquivo {
    private PosProcessador processador;

    public void setProcessador(PosProcessador processador){
        this.processador = processador;
    }

    public final void gerarArquivo(String nome,
```

```
                              Map<String, Object> propriedades)
            throws IOException {
        String conteudo = gerarConteudo(propriedades);
        byte[] bytes = conteudo.getBytes();
        bytes = processador.processar(bytes);
        FileOutputStream fileout = new FileOutputStream(nome);
        fileout.write(bytes);
        fileout.close();
    }

    protected abstract String
                gerarConteudo(Map<String, Object> propriedades);
}
```

Mas e o caso em que não há pós-processamento?

Na implementação original apresentada no capítulo anterior, o método `processar()` possuía uma implementação *default* para o caso de não haver nenhum pós-processamento. Esse caso deve ser tratado aqui nessa solução também! O que deve ser feito? Deve-se colocar um condicional para verificar se o processador foi configurado corretamente? Essa é uma solução possível, porém podemos utilizar o **Null Object** para solucionar esse problema.

Para fazer isso, uma implementação de `PosProcessador` chamada `NullProcessador` poderia ser criada com uma que retornasse o próprio array de bytes recebido como parâmetro sem nenhum pós-processamento. Sendo assim, para garantir que não ocorra um `NullPointerException` caso o conteúdo da variável `processador` seja nulo, uma instância dessa classe poderia ser atribuída a ela em algum momento da inicialização do objeto.

Esse exemplo mostra como é importante combinar as soluções dos padrões para chegar a uma estrutura final. Enxergue um padrão como um elemento que pode ser incluído para compor a sua solução e não a solução completa!

Entendendo o padrão Bridge

Essa solução caracteriza o padrão **Bridge**, que cria uma ponte entre duas hi-

erarquias ligadas por uma relação de composição permitindo que ambas variem de forma independente. Nesse caso, a ponte é caracterizada pela relação de composição entre a classe `GeradorArquivo` e a interface `PosProcessador`. Um cenário em que esse tipo de solução é comum é quando temos uma hierarquia de abstrações e outra com implementações, permitindo que cada uma possa variar independentemente. Neste caso, o `GeradorArquivo` e suas subclasses são uma abstração que caracteriza algoritmos de geração de arquivos em determinados formatos, porém ainda sem definir a implementação do pós-processamento.

A Figura 3.4 apresenta uma estrutura mais geral para o padrão **`Bridge`**. A classe `Abstracao` representa um conceito que precisa ser representado em um sistema e possui duas variabilidades independentes, ou seja, comportamentos não dependentes que podem ser estendidos e/ou modificados. A classe representada como `AbstracaoRefinada` especializa o comportamento da abstração através da implementação de seus *hook methods*. Já a abstração representada pela interface `Componente` compõe a classe `Abstracao` e representa uma de suas variabilidades. Dessa forma, quando essa operação precisar ser invocada em `Abstracao`, o método na classe que a compõe será chamado.

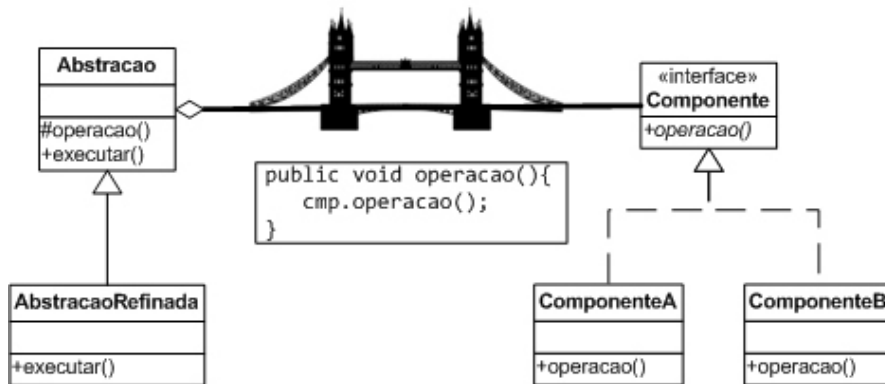



Figura 3.4: Estrutura Geral do Padrão Bridge

No exemplo apresentado, a classe `Abstracao` é representada pela classe `GeradorArquivo`, que representa uma abstração de como gerar arquivos a partir de mapas de propriedades. Seus refinamentos são classes que a especializam para que a geração dos arquivos seja feita em um formato específico, como XML e arquivos de propriedades. A interface `Componente` é representada pela interface

`PosProcessador` e fica claro a partir de suas implementações que ela representa uma operação de pós-processamento do arquivo.

Os Efeitos do Bridge

A principal motivação na implementação do padrão **`Bridge`** é tornar independentes os fatores que podem variar na solução. Qualquer subclasse de `Abstracao` pode ser facilmente composta com qualquer implementação da interface `Componente`, permitindo a combinação de ambas livremente sem duplicação de código. Caso, por exemplo, uma nova implementação de qualquer uma das duas hierarquias for criada, ela será facilmente combinada com o que já existe.

Um efeito colateral muito interessante é que as classes que foram separadas, também podem ser utilizadas em outro contexto. No exemplo, os algoritmos de pós-processamento poderiam ser utilizados na geração de outros tipos de arquivos ou mesmo para o envio de informações pela rede. Esse tipo de questão nos faz refletir se a princípio, pensando apenas nas responsabilidades, essa funcionalidade já não deveria ter sido atribuída a outra classe. Imagine acessar uma classe chamada `GeradorArquivo` para criptografar dados para serem enviados em uma rede. Dessa forma, uma consequência desse padrão também é o desacoplamento de responsabilidades, permitindo mais facilmente o seu reuso em outros contextos.

Um ponto que acho interessante no **`Bridge`** é o fato de sua solução utilizar ao mesmo tempo herança e composição. Escuto muitas pessoas dizerem para utilizar **sempre** a composição no lugar da herança, porém acho que a palavra **"sempre"** é muito complicada no contexto de design de software, pois não existe bala de prata nem uma solução mágica que irá resolver todos os seus problemas. A composição tem sim suas vantagens que serão apresentadas no presente capítulo, porém é importante sempre avaliar com cuidado o problema e os requisitos da solução para que se busque a mais apropriada.

O padrão não é o diagrama!

Um erro comum ao se aprender os primeiros padrões é achar que a implementação deve ser sempre exatamente como é mostrada no diagrama geral do padrão. O diagrama é apenas uma referência, pois a mesma solução pode ser implementada de diversas outras formas. Por exemplo, dependendo do contexto, `Implementacao` poderia ser implementada como uma classe abstrata ou a classe `Abstracao` poderia possuir mais de um *hook method* a ser implementado por suas subclasses. É por esse motivo, que ferramentas que geram código ou provêem uma estrutura de classes para a implementação de padrões são bastante criticadas por especialistas da área. É mais importante se preocupar em ter um solução adequada ao seu problema, do que seguir cegamente a estrutura do padrão.

3.3 Hook Classes

No capítulo anterior foi introduzido o conceito de *Hook Methods*, que são métodos que podem ser sobrescritos pelas subclasses como forma de estender e especializar o comportamento da classe. Nesse capítulo estamos vendo uma outra forma de alterar o comportamento das classes a partir da composição. Em vez de os pontos em que o comportamento pode variar serem definidos na mesma classe, eles são definidos em uma outra classe que compõe a classe principal. A essa classe que pode ser alterada damos o nome de *Hook Class*. A Figura 3.5 ilustra o conceito apresentado.

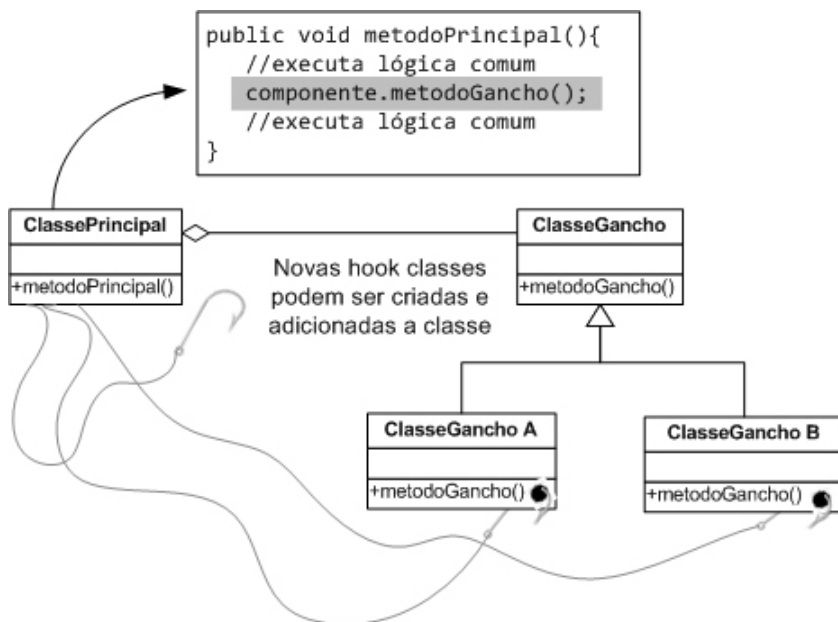



Figura 3.5: Representação do Conceito de Hook Classes

Da mesma forma que os *hook methods*, as *hook classes* são uma técnica utilizada pelos padrões para chegar a uma solução para um problema mais específico. O padrão **`Bridge`**, por exemplo, utiliza ao mesmo tempo um *hook method* e uma *hook class* como forma de separar dois pontos de extensão cujo comportamento pode variar de modo independente. Dessa maneira, é importante não apenas conhecer os padrões, mas compreender os princípios por trás de sua estrutura para entender melhor as consequências trazidas por uma decisão de design.

Diferenças Entre os Tipos de Hooks

Quando entendemos a ideia por trás dos *hook methods* e das *hook classes*, eles até que são bem parecidos. Em ambos os casos, a classe principal chama um método cuja implementação pode variar. No caso dos *hook methods* esse método está na mesma classe, podendo a implementação variar com a subclasse, e no caso das *hook classes* o método está em um objeto que compõe a classe, fazendo com que a implementação varie com a instância. Apesar de alguma semelhança, a utilização de uma técnica ou outra possui consequências bem diferentes.

A primeira delas está no momento em que a implementação que será utilizada

é escolhida. Quando um *hook method* é utilizado, a escolha é feita no momento em que o objeto é instanciado. Isso ocorre pois ao criarmos um novo objeto devemos escolher qual a classe concreta que será utilizada, e fazendo isso estamos escolhendo a sua implementação dos *hook methods*. Já no caso da utilização de *hook classes*, a implementação pode ser trocada a qualquer momento, bastando para isso trocar a instância que foi configurada. Observe que isso não é obrigatório, como no exemplo da classe `GeradorArquivo` apresentada nesse capítulo, que permitia a configuração da classe que a compunha apenas no construtor.

Nesse ponto, vale a pena retomar o exemplo do primeiro capítulo, no qual era necessário poder trocar a lógica que calculava o valor do estacionamento de acordo com o tempo e com o tipo de veículo. No cenário apresentado era extremamente importante poder trocar a implementação do cálculo após a criação do objeto `ContaEstacionamento`. Esse foi um dos motivos que fizeram a escolha do padrão **`Strategy`**, que utiliza uma *hook class*, ser adequado para a resolução do problema.

Outra questão que deve ser levada em consideração é o grau de dependência de diferentes pontos de extensão. Caso os *hook methods* sejam definidos na mesma classe, sendo eles na própria classe ou em uma *hook class*, as implementações ficam ligadas. Foi esse o problema apresentado no exemplo do `GeradorArquivo`, no qual pontos de extensão eram independentes mas ficavam definidos na mesma classe. Quando usamos apenas *hook methods*, pelo fato de não haver herança múltipla em Java, eles sempre ficam interligados. Ao utilizarmos *hook classes*, se colocarmos os métodos na mesma classe, o mesmo problema irá acontecer, porém se pusermos em classes diferentes, cada implementação poderá ser configurada de forma independente. O padrão **`Bridge`** aborda justamente essa questão, utilizando diferentes estratégias de extensão para que as implementações possam variar de forma independente.

Evoluindo o Modelo de Classes

Em um artigo clássico sobre padrões para evolução de frameworks [8], é mostrado que normalmente os pontos de extensão são primeiramente definidos utilizando herança, o que é conhecido como *whitebox framework*, para só em seguida evoluir para o uso de composição, chamado de *blackbox framework*. Por mais que a composição pareça ser mais vantajosa que a herança por suas características estruturais, é muito mais fácil deixar sua classe aberta à extensão através de herança.

O termo *whitebox framework* se refere a frameworks cujas classes permitem sua

extensão a partir da herança. Devido ao fato das subclasses terem acesso a detalhes internos da classe que está sendo estendida, sua estrutura fica exposta. O termo *whitebox*, ou caixa branca, é uma metáfora a essa visibilidade que o desenvolvedor precisa ter de como as coisas são na classe internamente. Com a utilização desse paradigma, qualquer método público ou protegido que não seja final, é um potencial *hook method* a ser especializado pelas subclasses. Dessa forma, quando não se sabe exatamente o que será necessário ser estendido, a herança é uma boa alternativa por possibilitar que vários pontos fiquem em aberto. Uma desvantagem dessa abordagem é que quem desenvolver a classe que faz a extensão deve conhecer a estrutura interna da superclasse, tornando a tarefa mais complexa.

Por outro lado, o termo *blackbox framework* se refere a frameworks cujo mecanismo de extensão se baseia em composição. Nesse caso, o desenvolvedor deve criar implementações que obedecem a uma interface fornecida e utilizá-las para compor a classe principal. O termo *blackbox*, ou caixa preta, é utilizado porque o desenvolvedor não precisa conhecer a parte interna das classes do framework para estendê-las. Apesar de possuir as vantagens da composição, esse tipo de ponto de extensão é mais difícil de ser identificado, pois é preciso saber a porção exata da funcionalidade que será delegada. Em compensação, depois que esses pontos forem identificados, é mais fácil para outros desenvolvedores alterarem o comportamento, pois eles só precisam entender as interfaces que estão implementando.

Na verdade, nada precisa ser completamente preto ou branco. O caminho natural é, ao criar a primeira classe, deixar alguns possíveis *hook methods* em aberto para que seja possível realizar a extensão do comportamento. À medida que as extensões forem sendo criadas e os principais pontos onde o comportamento é especializado com frequência forem identificados, a estrutura pode ser refatorada para o uso de *hook classes*.

3.4 State - Variando o Comportamento com o Estado da Classe

"Eu costumava partir por semanas em um estado de confusão."
– Albert Einstein

Um cenário bastante comum ao desenvolvermos um software é uma determinada entidade mudar de estado durante o decorrer de sua execução, alterando seu comportamento de acordo com esse estado. Uma conta corrente em um banco, por

exemplo, comporta-se diferente caso o saldo esteja negativo. Em um jogo, as mesmas ações podem gerar diferentes efeitos em um personagem quando ele está em estados diferentes, como quando pega algum item ou está dentro d'água.

A implementação comum nesses casos é criar uma série de condicionais em que, em diversos pontos da classe, verifica-se o estado do objeto e executa-se a lógica mais adequada. Seguindo essa ideia, alguns estados são suficientes para deixar o código bastante confuso. É comum que a mesma sequência de condicionais seja encontrada em diversos pontos do código onde o estado do objeto possui influência. Além disso, essa estrutura torna complicada a adição de mais estados, pois isso demandará alterações no código da própria classe e irá aumentar ainda mais a quantidade de condicionais. Como resolver isso? Conheça o padrão **`State`**!

A Estrutura do State

Essa seção irá apresentar o padrão **`State`**, o qual utiliza composição para permitir essa variação de comportamento de acordo como o estado de uma entidade do sistema. Nesse padrão, os estados são representados a partir de classes que obedecem a uma abstração comum, podendo ser uma interface ou uma superclasse. Dessa forma, a entidade é composta por um objeto do tipo dessa abstração e, nos métodos de negócio, delega para ele o comportamento de toda parte que depende do seu estado. A Figura 3.6 representa a estrutura desse padrão.

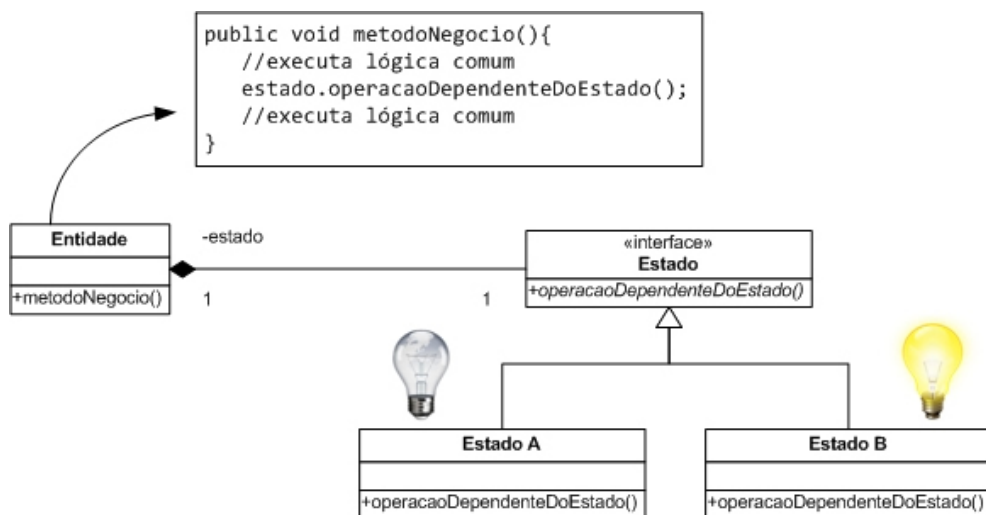



Figura 3.6: Estrutura do padrão State

Sendo assim, apenas a lógica comum será mantida na entidade e o comportamento do objeto específico de cada estado estará definido em cada subclasse. Quando o estado for alterado, basta trocar a instância utilizada para caracterizar o estado, que consequentemente o comportamento da entidade será alterado. Nesse caso, o estado é uma *hook class* para a qual está sendo delegado. Uma consequência desse padrão é a simplicidade em alterar estados existentes, ou mesmo inserir novos estados no ciclo de vida da entidade. A desvantagem é que, devido à divisão da lógica dos estados, fica mais difícil ter uma visão global dos estados e transições que podem ser realizadas.

Busca em Profundidade

Calma! Você não está em um livro de algoritmos! Acho muito interessante quando é possível resolver um problema clássico na área de projeto de algoritmos e estrutura de dados utilizando algum padrão. Isso nos mostra que eles são úteis para qualquer tipo de software. O algoritmo abordado nessa seção é a busca em profundidade em grafos. Nesse algoritmo, à medida que os nós do grafo vão sendo percorridos, eles mudam de cor e o comportamento do algoritmo depende da cor do nó que está sendo percorrido. Dessa forma, o `State` será utilizado para configurar o comportamento de cada nó de acordo com seu estado.

A busca em profundidade é um algoritmo de busca de grafos, que se inicia com um nó raiz e vai se explorando cada um de seus ramos, sempre na maior profundidade possível, antes de retroceder e processar os outros ramos. Para quem não se lembra do algoritmo de busca em largura em grafos, segue uma breve explicação de como ele funciona. A execução inicia-se a partir de um nó e todos iniciam-se com a cor branca. Quando um nó começa a ser processado, ele adquire a cor cinza que significa que ele está sendo processado. Ao adquirir a cor cinza, são processados todos os seus nós adjacentes que tenham a cor branca. Ao finalizar a execução o nó passa da cor cinza para cor preta, que indica que seu processamento foi concluído. O algoritmo termina quando o nó raiz onde o algoritmo iniciou adquire a cor preta.

A Figura 3.7 apresenta um exemplo de execução do algoritmo passo a passo em um grafo. Nesse exemplo, conforme os nós vão mudando de estado, eles adquirem uma cor diferente e as arestas que já tiverem sido percorridas são colocadas em negrito. A partir do exemplo é possível entender de uma forma mais concreta como os nós são percorridos e a situação em que mudam de estado.

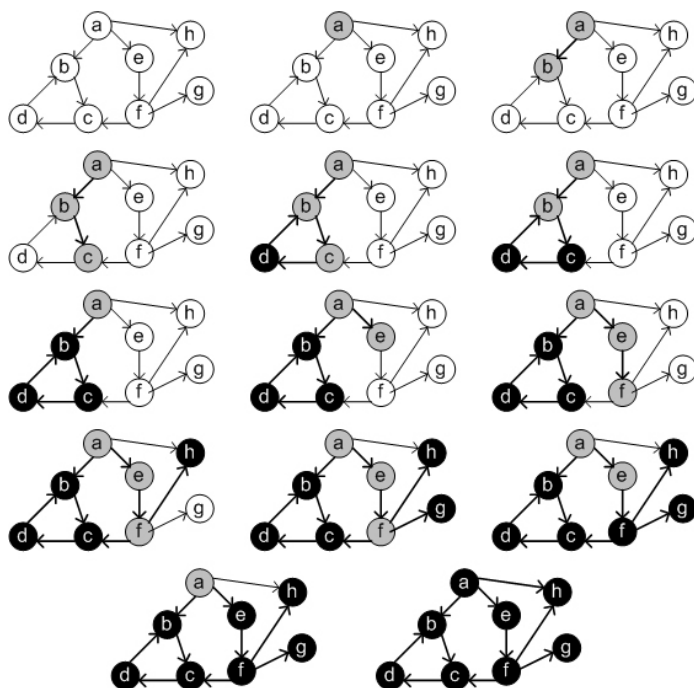



Figura 3.7: Busca em Profundidade Passo a Passo

Implementação da Busca em Profundidade Utilizando State

O primeiro passo da implementação é definir um nó e quais os pontos da execução que diferem com o estado. No contexto do algoritmo, as informações relevantes de um nó são o seu nome, seus nós adjacentes e sua cor. Essa classe possui um método chamado buscaProfundidade() que irá efetivamente executar o algoritmo. Os pontos identificados que podem variar com o estado são: (a) a própria execução da busca em profundidade, pois no caso do nó ser preto ou cinza ela não deve ser executada; e (b) quando o nó assume uma cor. A implementação realizada para classe `No` está representada na listagem a seguir.

Listagem 3.2 - Definição da classe que representa um nó:

```
public class No {
    private Set<No> adjacentes = new HashSet<>();
    private Cor cor;
    private String name;

    public No(String name){
        this.name = name;
        cor = new Branco();
    }
    public void buscaProfundidade(List<No> list){
        cor.busca(this, list);
    }
    public Set<No> getAdjacentes() {
        return adjacentes;
    }
    public void addAdjacentes(No adj) {
        adjacentes.add(adj);
    }
    public void setCor(Cor cor, List<No> list) {
        this.cor = cor;
        cor.assumiu(this ,list);
    }
    public String toString() {
        return name;
    }
}
```

O atributo `cor` representa o estado do nó na execução do algoritmo. Ob-

serve que métodos são invocados no objeto desse atributo quando o método `buscaProfundidade()` é invocado e quando uma nova cor é configurada a partir do método `setCor()`. O `buscaProfundidade()` recebe uma lista que deve ser populada com os nós do grafo na ordem que terminaram de ser executados pelo algoritmo.

A listagem a seguir apresenta a classe abstrata `Cor` e suas três subclasses. Observe que cada cor implementa os *hook methods* da definição do estado, de acordo com a descrição do algoritmo. Quando o nó no estado `Branco` recebe a chamada da busca, o mesmo deve passar para a cor `Cinza`. Essa, por sua vez, ao ser assumida pelo nó inicia o processamento de todos os nós adjacentes, assumindo a cor `Preto` ao seu final. Finalmente, ao se tornar `Preto`, o nó deve ser adicionado na lista recebida como parâmetro pelo algoritmo.

Listagem 3.3 - Implementação dos estados de um nó:

```
//Classe que abstrai o estado de um nó
public abstract class Cor {
    void busca(No no, List<No> list){}
    void assumiu(No no, List<No> list){}
}

//Implementações dos três estados possíveis
public class Branco extends Cor {
    public void busca(No no, List<No> list) {
        no.setCor(new Cinza(), list);
    }
}
public class Cinza extends Cor {
    void assumiu(No no, List<No> list) {
        for(No adj : no.getAdjacentes())
            adj.buscaProfundidade(list);
        no.setCor(new Preto(), list);
    }
}
public class Preto extends Cor {
    void assumiu(No no, List<No> list) {
        list.add(no);
    }
}
```

A listagem a seguir mostra a criação do grafo apresentado na Figura 3.7 e a execução do algoritmo. Ao seu final, os nós são impressos na ordem em que seu processamento foi finalizado.

Listagem 3.4 - Execução do algoritmo para o grafo do exemplo:

```
public static void main(String[] args) {
    No a = new No("A");     No b = new No("B");
    No c = new No("C");     No d = new No("D");
    No e = new No("E");     No f = new No("F");
    No g = new No("G");     No h = new No("H");

    a.addAdjacentes(b);     b.addAdjacentes(c);
    c.addAdjacentes(d);     d.addAdjacentes(b);
    a.addAdjacentes(e);     e.addAdjacentes(f);
    f.addAdjacentes(c);     f.addAdjacentes(g);
    f.addAdjacentes(h);     a.addAdjacentes(h);

    List<No> l = new ArrayList<>();
    a.buscaProfundidade(l);
    for(No n : l)
        System.out.println(n);
}
```

Nesse algoritmo, diversas ações dependem do estado do nó, como se ele deve ser adicionado na lista ou se os nós adjacentes devem ser processados. Apesar disso, observe que nenhum comando condicional foi utilizado na implementação!

Utilizando um Enum para Implementar o State

Quando temos que representar estados de um objeto em um software, frequentemente utilizamos enumerações. O problema é que muitas vezes as enumerações possuem apenas a definição dos estados, ficando a lógica específica de cada estado presa dentro dos condicionais na classe. Construções do tipo `enum` podem definir métodos, inclusive abstratos, que podem ser sobrescritos na definição de cada estado. Segue na listagem a seguir como as cores dos nós do grafo poderiam ser definidas utilizando uma enumeração. Em relação ao resto do código, somente a inicialização do atributo `cor` da classe `No` precisaria ser modificada.

Listagem 3.5 - Execução do algoritmo para o grafo do exemplo:

```
public enum Cor {

    BRANCO{
        public void busca(No no, List<No> list) {
            no.setCor(CINZA, list);
        }
    },CINZA{
        void assumiu(No no, List<No> list) {
            for(No adj : no.getAdjacentes())
                adj.buscaProfundidade(list);
            no.setCor(PRETO, list);
        }
    },
    PRETO{
        void assumiu(No no, List<No> list) {
            list.add(no);
        }
    };

    void busca(No no, List<No> list){}
    void assumiu(No no, List<No> list){}
}
```

Porém existem algumas situações em que a utilização de enumerações é desaconselhada na representação de um estado. A primeira é quando é desejável que o estado seja um ponto de extensão e que novos estados possam ser definidos. Nesse caso, como os possíveis estados são definidos dentro de uma enumeração fixa, não se pode adicionar um novo sem a modificação do código do próprio enum. Outra situação é quando algum estado precisa armazenar uma informação específica do objeto que está sendo composto por ele. Nesse caso, como cada instância do enum é compartilhada por todos que a possuem, a informação não poderia ser diferente para cada uma.

3.5 Substituindo Condicionais por Polimorfismo

No exemplo apresentado, os nós já possuíam um estado explícito segundo a descrição do algoritmo, porém um cenário para utilização do padrão `State` muitas vezes é difícil de ser identificado inicialmente. O conceito do que significa um estado para uma determinada entidade do software pode começar a ficar explícito somente no

momento da codificação. A repetição de condicionais similares em diversos pontos da mesma classe pode ser um sinal de que seria adequada a refatoração do código em direção ao padrão `State`.

O mesmo vale para outros padrões que utilizam composição. Um exemplo é quando uma classe possuir um método grande que utiliza condicionais para selecionar dentre alternativas de implementação para um mesmo algoritmo. Nesse caso, a refatoração poderia ser na direção do padrão `Strategy`. O exemplo apresentado na introdução do livro ilustra bem uma refatoração em direção à composição.

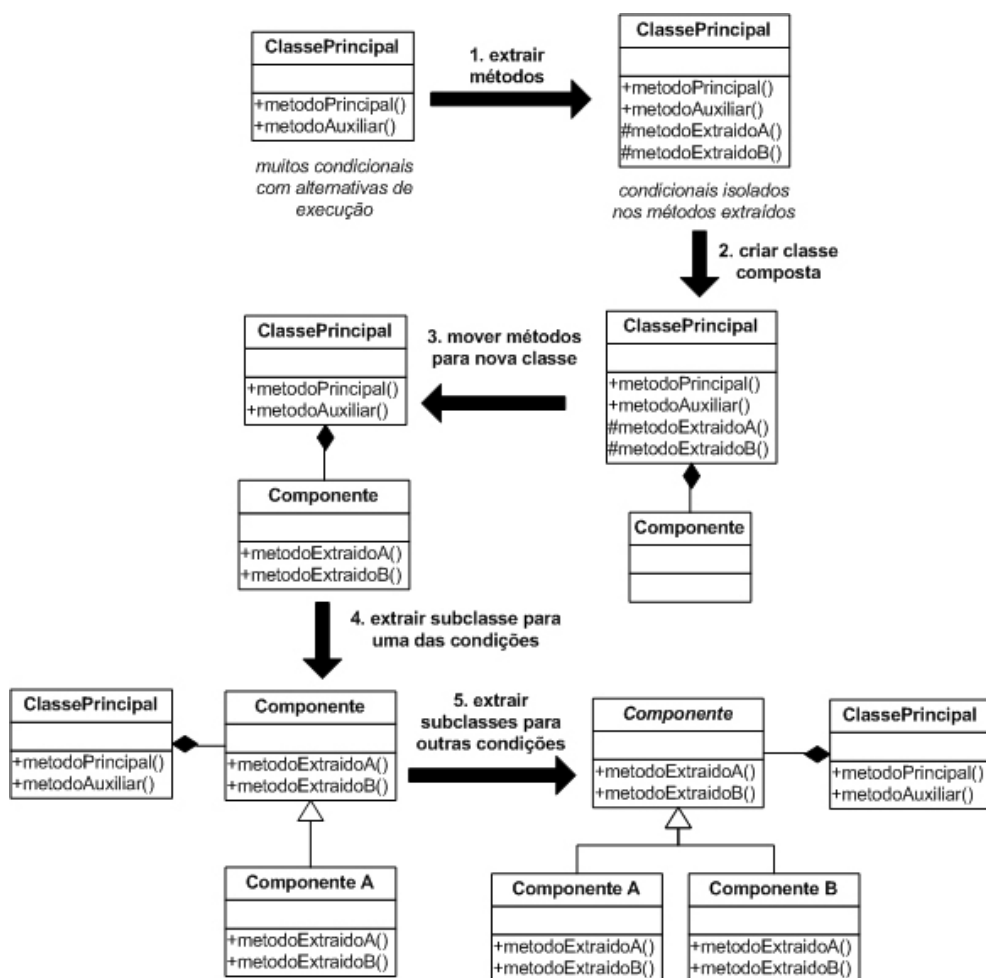



Figura 3.8: Refatorando em Direção a Composição

A Figura 3.8 apresenta o processo que pode ser utilizado para extrair a classe que será utilizada para a composição e para a criação das implementações de cada alternativa. O primeiro passo consiste na identificação dos locais com a lógica condicional como descrito acima e na sua extração para métodos que a princípio ficarão na própria classe. Em seguida, uma nova classe deve ser criada e um atributo do tipo dela deve ser introduzido na classe principal que se deseja refatorar. Note que a princípio ela estará vazia e nenhum uso dela é feito a partir da principal.

Com a nova classe criada, o próximo passo é mover os métodos extraídos na primeira etapa da refatoração para ela. Cada método deve ser passado para a nova classe como método público e os locais onde era invocado devem agora chamá-lo a partir do atributo criado. É aconselhável que cada método seja migrado separadamente, executando testes em cada passo. Uma questão que deve ser considerada nesse ponto é relacionada aos dados que são utilizados pela porção da lógica que está indo para outra classe. Quando a informação for um parâmetro relacionado com o algoritmo em si, normalmente ele é incluído como um atributo da classe que está sendo extraída. Quando é uma informação da classe principal, normalmente ele é passado como parâmetro, ou a própria instância da classe é passada como parâmetro.

Depois que a lógica condicional for migrada para outra classe, é possível começar a extrair subclasses com cada uma das implementações. O conteudo de um condicional na superclasse deve ser movido para uma subclasse. Adicionalmente, é preciso atribuir uma instância dessa classe na principal quando ela precisar ser invocada. Essa troca de instância é realizada normalmente no momento em que a variável utilizada nos condicionais é alterada. Por exemplo, se a refatoração for para o padrão **`State`**, então a mudança da instância deve ocorrer no momento da mudança de estado. Com isso, a lógica que antes era selecionada com condicional agora será invocada através de polimorfismo. As duas listagens a seguir apresentam respectivamente o antes e o depois da criação de uma nova subclasse.

Listagem 3.6 - Código do componente antes da refatoração:

```
public class Componente {
    public void metodoExtraido() {
        if (possibilidadeA) {
            //lógica referente ao cenário A
        } else if(possibilidadeB) {
            //lógica referente ao cenário B
        } else if(possibilidadeC) {
            //lógica referente ao cenário C
        }
```

```
    }
}
```

Listagem 3.7 - Código do componente e sua subclasse depois da refatoração:

```
public class Componente {
    public void metodoExtraido() {
        if (possibilidadeB) {
            //lógica referente ao cenário B
        } else if(possibilidadeC) {
            //lógica referente ao cenário C
        }
    }
}

public class ComponenteA extends Componente {
    @Override
    public void metodoExtraido() {
        //lógica referente ao cenário A
    }
}
```

Depois que uma subclasse for extraída, a refatoração deve ser concluída com a extração das outras subclasses. É importante não tentar criar todas as subclasses de uma só vez. A cada subclasse extraída, deve-se executar os testes para verificar ser ela foi realizada corretamente. Ao finalizar a extração, a superclasse pode ficar com algum comportamento *default*, caso exista, ou os métodos podem ser transformados em abstratos caso toda lógica tenha sido movida.

3.6 Compondo com Múltiplos Objetos - Observer

"A visão de um observador imparcial é uma janela para o mundo."
– Kenneth L. Pike

Nos exemplos vistos até o momento, a composição ocorre apenas com uma instância. Nessa seção será apresentado um padrão que utiliza a composição com múltiplos objetos: o `Observer`. Esse é um padrão muito importante e que frequentemente é necessário em aplicações. É aplicável quando existe um objeto cujos eventos precisam ser observados por outros objetos. Um exemplo seria um componente

gráfico cujos eventos precisam ser tratados pela aplicação. Um outro exemplo, talvez menos óbvio, é um objeto de dados cujas mudanças demandam mudanças em outras partes do sistema.

A próxima seção apresenta um exemplo do padrão **Observer**, em que mudanças em um objeto são notificadas a outros objetos interessados. Em seguida esse padrão será analisado de forma mais profunda, avaliando sua estrutura básica e suas consequências. Por fim, serão mostradas algumas APIs existentes que fazem uso dele.

Notificando mudanças em uma Carteira de Ações

O exemplo que será abordado envolve um objeto que representa uma carteira de ações. Essa carteira possui um mapa com os códigos das ações e sua respectiva quantidade. De acordo com as ações do usuário ou com gatilhos configurados na aplicação, ações podem ser compradas ou vendidas alterando sua quantidade. A questão é que existem diversas classes interessadas em saber quando uma informação é alterada nessa classe para poder executar a sua lógica. Por exemplo, um componente que exibe um gráfico com a quantidade de cada ação da carteira precisa saber quando houve uma mudança para ser atualizado. Outro exemplo seria um componente que fizesse um log das alterações realizadas. Poderia também existir um componente para fazer a auditoria dos dados.

Caso as notificações sejam feitas individualmente para cada necessidade, a classe que representa a carteira de ações ficaria acoplada às classes que precisaria notificar. Isso não é desejável. Além de dificultar mudanças nas classes envolvidas, tornaria complicado a adição de novas classes interessadas em mudanças nessas informações.

Para resolver esse problema, o padrão **Observer** será implementado na solução. Será criada uma interface `Observador`, apresentada na listagem a seguir, a qual deve ser implementada pelas classes que desejam receber os eventos de mudança na quantidade das ações. Implementações dessa interface poderão ser registradas na classe `CarteiraAcoes`, cuja listagem também está apresentada, através do método `addObservador()`. Dessa forma, todos os observadores registrados receberão uma notificação quando uma mudança for realizada. Observe que o método `notificar()` chama o método `mudancaQuantidade()` da interface `Observador` em todas as instâncias registradas.

Listagem 3.8 - Interface para receber notificações da carteira da ações:

```
public interface Observador {
```

```
    void mudancaQuantidade(String acao, Integer qtd);
}
```

Listagem 3.9 - Classe cujos dados podem ser observados por outras classes:

```
public class CarteiraAcoes {
    private Map<String,Integer> acoes = new HashMap<>();
    private List<Observador> obs = new ArrayList<>();

    public void adicionaAcoes(String acao, Integer qtd) {
        if(acoes.containsKey(acao)){
            qtd += acoes.get(acao);
        }
        acoes.put(acao, qtd);
        notificar(acao, qtd);
    }

    private void notificar(String acao, Integer qtd) {
        for(Observador o: obs)
            o.mudancaQuantidade(acao, qtd);
    }

    public void addObservador(Observador o) {
        obs.add(o);
    }
}
```

A seguir, são apresentadas duas listagens de classes que implementam a interface `Observador`: `AcoesLogger` e `GraficoBarras`. A classe `AcoesLogger` possui uma implementação bem simples e apenas registra a nova quantidade recebida no console. Já a classe `GraficoBarras` utiliza o componente de geração de gráficos JFreeChart para a criação de uma janela com um gráfico de barras que é atualizado toda vez que uma mudança é notificada.

Listagem 3.10 - Classe que observa mudanças nas quantidades e registra no console:

```
public class AcoesLogger implements Observador {
    public void mudancaQuantidade(String acao, Integer qtd) {
        System.out.println("Alterada a quantidade da ação "
                + acao + " para " + qtd);
```

```
    }
}
```

Listagem 3.11 - Classe que coloca as quantidades em um gráfico de barras:

```
public class GraficoBarras implements Observador {

    private DefaultCategoryDataset dataset;
    private JFrame frame = new JFrame();

    public GraficoBarras() {
         dataset = new DefaultCategoryDataset();
         JFreeChart chart = ChartFactory.createBarChart(
                 "Carteira de Ações","Siglas",
                 "Quantidade", dataset, PlotOrientation.VERTICAL,
                    false, true, false);
         ChartPanel chartPanel = new ChartPanel(chart);
         frame.setContentPane(chartPanel);
         frame.setSize(500, 270);
         frame.setVisible(true);
         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    public void mudancaQuantidade(String acao, Integer qtd) {
        dataset.setValue(qtd, "Quantidade", acao);
    }
}
```

Utilizando o JFreeChart

Se você deseja executar o exemplo apresentado, precisa baixar o JFreeChart no endereço *http://www.jfree.org/jfreechart/* e adicionar os arquivos jfreechart-1.x.x.jar e jcommon-1.x.x.jar no classpath. Os gráficos desse framework sempre são divididos em duas partes: o conjunto de dados e a sua representação visual. No exemplo apresentado, o atributo `dataset` da classe `DefaultCategoryDataset` representa o conjunto de dados e a variável local `chart` representa a parte visual. No caso, com a alteração do conjunto de dados associado ao gráfico, a representação também é atualizada.

Para finalizar o exemplo, a listagem a seguir apresenta um método `main()` que associa os observadores na carteira de ações e adiciona ações a carteira. No início do código, as instâncias de `GraficoBarras` e `AcoesLogger` são criadas e associadas ao objeto da classe `CarteiraAcoes`. Em seguida, o método `Thread.sleep()` é utilizado para gerar um intervalo entre cada quantidade inserida na carteira de ações, para ser possível ver de forma progressiva as alterações no gráfico e as modificações registradas no console. A Figura 3.9 apresenta o gráfico gerado no final da execução.

Listagem 3.12 - Exemplo de execução da carteira de ações com observadores:

```
public static void main(String[] args) throws Exception {
    GraficoBarras gb = new GraficoBarras();
    AcoesLogger al = new AcoesLogger();
    CarteiraAcoes c = new CarteiraAcoes();
    c.addObservador(gb);
    c.addObservador(al);

    Thread.sleep(2000);
    c.adicionaAcoes("COMP02", 200);
    Thread.sleep(2000);
    c.adicionaAcoes("EMP34", 400);
    Thread.sleep(2000);
    c.adicionaAcoes("ACDF89", 300);
    Thread.sleep(2000);
    c.adicionaAcoes("EMP34", -200);
    Thread.sleep(2000);
    c.adicionaAcoes("COMP02", 150);
}
```

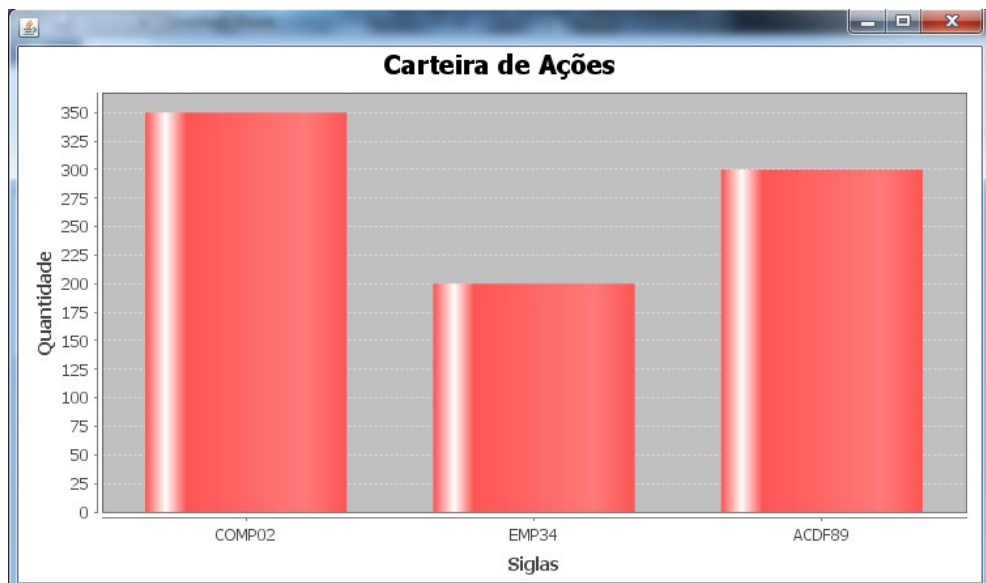



Figura 3.9: Exemplo do Gráfico Gerado pela Aplicação

Indo a fundo no Padrão Observer

A estrutura do padrão `Observer` é bem simples: a classe que está sendo observada é composta por uma lista de observadores. Além disso, a classe normalmente oferece métodos que permitem a adição e remoção desses observadores em tempo de execução. Dessa forma, quando o estado da classe observada é alterado, todos os observadores são notificados. A Figura 3.10 apresenta um diagrama com a estrutura do padrão.

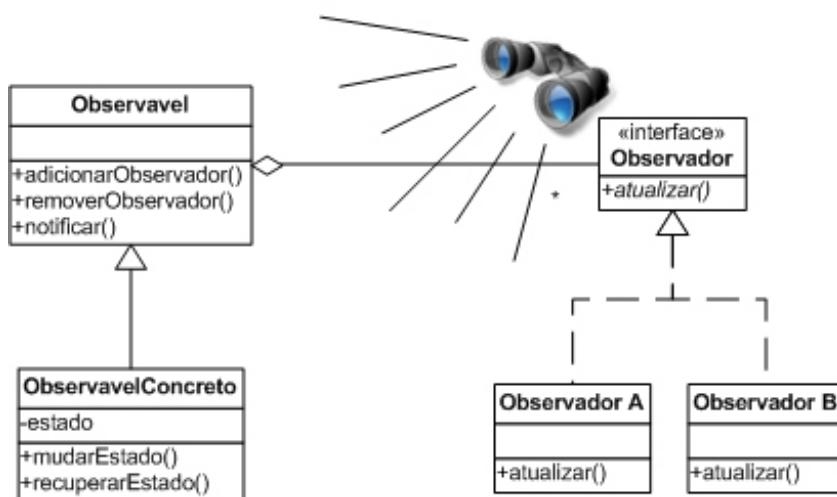



Figura 3.10: Estrutura do Padrão Observer

Observe que no diagrama existe uma hierarquia de classes composta pelas classes `Observavel` e `ObservavelConcreto` que não havia sido representada no exemplo, no qual esses dois papéis são desempenhados pela mesma classe. Essa abstração de objetos observáveis pode ser extremamente útil quando houver mais de uma classe que puder emitir notificações. Imagine, por exemplo, diversos componentes gráficos que podem notificar a ocorrência de eventos gerados por ações do usuário. Nesse contexto pode ser importante a capacidade de abstrair esse comportamento e atribuir observadores ao mesmo tipo de classe.

Uma importante consequência desse padrão é o desacoplamento entre a classe que está sendo observada e a observadora. A interface que deve ser implementada pelos observadores é o contrato que une as implementações. Dessa forma, um mesmo observador pode ser adicionado em diferentes objetos, recebendo notificações de todos eles, e vários observadores diferentes podem receber notificações do mesmo objeto. No **`Observer`** ao invés de possuir uma *hook class*, a classe observada pode possuir um conjunto, sendo que o *hook method* de todas é chamado no momento em que deve ocorrer a notificação.

É importante ressaltar, especialmente ao se falar do **`Observer`**, que diversas variações são possíveis. Por exemplo, podem existir vários métodos de notificação para diferentes tipos de evento, que precisam ser implementados pela classe observadora. Nesse caso, cada método seria chamado em uma situação diferente. No exemplo

apresentado, poderia haver um método para notificar quando uma ação é removida da carteira. Outro ponto em que pode haver variação é em como os observadores são gerenciados. Eles poderiam, por exemplo, ser passados no construtor e, outras vezes, não faz sentido haver um método de exclusão. Lembre-se que o padrão é apenas uma referência e cabe ao desenvolvedor adaptá-lo ao seu contexto!

Ao lidar com a composição de diversas *hook classes*, é importante levar em consideração como o que ocorre em uma influencia as outras. Por exemplo, o que acontece se uma das classes lança uma exceção? As outras devem continuar sendo executadas? Em caso positivo, deve haver um tratamento de erro separado para cada invocação. Outra questão seria em relação à possibilidade de demora na execução de um observador influenciar a velocidade de notificação dos outros. Dependendo dos requisitos da aplicação, pode ser necessário a chamada do *hook method* de cada classe em um *thread* diferente.

Padrão Observer nas APIs Java

O padrão **`Observer`** pode ser visto com frequência em várias APIs da linguagem Java. Um dos motivos dele poder ser notado de forma tão explícita é porque diversas classes permitem que suas mudanças sejam recebidas pela aplicação através de observadores. Como foi dito, essa é uma boa forma de permitir o desacoplamento entre a classe que gera e a que recebe o evento.

A interface `ActionListener`, por exemplo, é utilizada no Swing nas classes que desejam tratar eventos de interface. No caso, os objetos observados são componentes de interface, como a classe `JButton`. O observador, que também é chamado *listener*, é adicionado no componente através do método `addActionListener()` e precisa implementar o método `actionPerformed()` para tratar os eventos.

Outro exemplo de implementação do **`Observer`**, agora nas APIs do Java EE, é o `MessageListener` para o recebimento de mensagens JMS. Essa interface precisa ser implementada por classes que desejam receber mensagens de um serviço de mensageria, as quais devem implementar o método `onMessage()` para tratar cada mensagem recebida. Uma instância de `MessageConsumer` com a fila ou o tópico de origem é onde o *listener* deve ser configurado. Os Message Driven Beans, que são EJBs responsáveis por tratar mensagens JMS, também implementam essa interface.

Há dezenas de outros exemplos dentro das clássicas APIs do Java. Todos os listeners de sessão das servlets, os listeners de fase da JSF e os listeners de mudanças de valores em entidades na JPA são bons exemplos de implementação do **`Observer`**, que desacoplam seu código e facilitam a extensão. Todos os observadores podem

trabalhar de forma totalmente independente, sem ter conhecimento da existência dos outros.

Uma coisa interessante em aprender padrões é que a partir deles fica mais fácil compreender o funcionamento de APIs e frameworks que utilizamos. É como se passássemos a olhar para eles com outros olhos. Imagino que seja algo comparável a quando estudamos física, química ou biologia e passamos a compreender melhor o mundo ao nosso redor. Ao perceber que um padrão foi implementado em uma solução, é mais fácil entender com utilizar aquela API e quais as consequências do uso daquela solução naquele contexto.

A Interface Observer e a Classe Observable

Uma curiosidade que poucos sabem é que existe desde a JDK 1.0 a interface `java.util.Observer` e a classe `java.util.Observable`. A interface `Observer` possui o método `update()` que recebe como parâmetro a instância de `Observable` que gerou o evento e mais um objeto que é o parâmetro passado para o método `notifyObservers()`. A ideia da classe `Observable` é que ela seja estendida por classes que queiram emitir notificações sobre mudanças em seu estado. Ela possui métodos implementados para gerenciar e notificar as instâncias de `Observer`.

O que é mais interessante é que essa classe não é utilizada em nenhum lugar da API da JDK, mesmo o padrão **Observer** sendo utilizado em diversos contextos. A questão principal é que essas classes consideram o **Observer** como uma implementação pronta e não como uma solução que pode ser adaptada para diversos contextos. Por mais que a solução apresentada por essas classes seja reutilizável, dificilmente ela será mais adequada do que uma implementação do padrão para um contexto específico.

3.7 Considerações Finais do Capítulo

Essa capítulo apresentou como a composição pode ser utilizada para inserir pontos de extensão que permitem que uma porção de lógica possa ser inserida durante a execução de uma classe. Esses pontos podem ser usados para inserir uma lógica específica da aplicação ou de um domínio em uma classe mais genérica, tornado-a mais

reutilizável. Por meio de *hook classes*, a adaptação de comportamento é feita através da combinação de objetos, podendo o desenvolvedor criar novos componentes para a configuração de classes final.

A extensão de comportamento pode ocorrer por diferentes motivos e em variados contextos. Pode ser delegada para outra classe a execução de um algoritmo que pode ser trocado, como é o caso do padrão **`Strategy`**. Outro cenário é a delegação da lógica relativa às diferenças de estado de um objeto, sendo esse o foco do padrão **`State`**. Finalmente, o padrão **`Observer`** associa a execução de métodos de diversas outras classes à mudança de estado ou a eventos ocorridos em um objeto.

Além de compreender os padrões e esses cenários que favorecem o uso da composição, é importante também conhecer os princípios e consequências por trás dessas decisões. A composição pode inclusive ser combinada com outros tipos de solução, como no padrão **`Bridge`** no qual ela é combinada com o uso de herança em uma hierarquia diferente.

Capítulo 4

Composição Recursiva

"Recursão: ver recursividade. Recursividade: se ainda não entendeu, ver recursão"
– Piada Nerd

No capítulo anterior foi visto como a composição pode ser utilizada para combinar classes de diferentes tipos para permitir que a funcionalidade seja estendida. Através do polimorfismo, a classe composta pode abstrair a implementação da classe que a está compondo, permitindo que diversas classes diferentes possam ser colocadas naquela posição. Agora imagine se uma classe puder ser composta pela sua própria abstração, ou, em outras palavras, se ela possuir um atributo de sua superclasse ou de uma de suas interfaces. Isso permitiria que uma instância da mesma classe pudesse ser utilizada para compô-la, tornando possível a criação de uma grande estrutura dessa forma.

Eu sei que isso parece ser muito abstrato, mas vamos tentar pensar em um exemplo do mundo físico para ilustrar esse conceito. Imagine uma peça de Lego que provê na parte de cima uma forma para que outras peças possam ser encaixadas e, na parte de baixo, a forma do encaixe para que ela seja ligada na parte de cima. Como a

mesma peça possui as duas formas de encaixe, possível ligar e combinar várias peças do mesmo tipo em diferentes configurações. Quando consideramos que podem existir peças diferentes com o mesmo formato de encaixe, as possibilidades são ainda maiores.

Voltando ao mundo do software, é como se a forma que uma peça provê em sua parte superior fosse a interface da classe. Analogamente, o tipo de um atributo seria o encaixe disponibilizado para que uma outra peça possa se encaixar. Dessa forma, uma classe que implementa uma interface se encaixa em um atributo daquele tipo. Na composição simples, utilizando essa ideia é possível conectar apenas dois objetos. Mas quando uma classe provê uma interface e um atributo do mesmo tipo, é possível criar uma estrutura com diversos objetos. Nesse caso, as próprias possibilidades de configurações dessa estrutura já tornam o software mais flexível.

4.1 Compondo um Objeto com sua Abstração

A recursividade é um conceito amplamente utilizado em programação. Uma função recursiva, por exemplo, é aquela que chama a si mesma como parte da sua lógica. Em um algoritmo recursivo, a função é chamada novamente para diminuir o problema que precisa ser resolvido. Dessa forma, ele vai sendo reduzido até o ponto em que sua resolução é trivial. Esse é chamado de ponto de parada.

Um exemplo clássico e simples de função recursiva é o fatorial de um número, que consiste na multiplicação de todos os números inteiros e positivos menores ou iguais àquele número. A listagem a seguir apresenta a implementação do fatorial utilizando recursão. Observe que, pela definição de fatorial, o fatorial de um número `n` é `n` vezes o fatorial de `n-1`. O fato de que o fatorial de 1 é 1 é utilizado como critério de parada.

Listagem 4.1 - Cálculo do fatorial com algoritmo recursivo:

```
public static int fatorial(int n){
    if(n == 1)
        return 1;
    return n * fatorial(n-1);
}
```

Quando criamos uma estrutura com objetos, é possível utilizar também o conceito de recursividade para que ela seja definida utilizando sua própria definição. Isso é feito quando uma classe contém um atributo do próprio tipo da classe. Esse

tipo de estrutura é muito utilizado para definir muitas estruturas de dados, como listas ligadas, árvores e grafos. Por exemplo, no Capítulo 3 foi apresentada busca em profundidade de grafos, onde a classe No possuía uma lista de instâncias de No.

Para exemplificar vamos considerar o código do elemento de uma lista ligada definido a seguir. A classe `Elemento` possui um atributo `proximo` que possui o mesmo tipo. Dessa forma, é possível encadear diversas instâncias dessa classe formando uma lista. O método `contar()` é um método recursivo no qual o próprio método é chamado novamente, porém na instância que estiver no atributo `proximo`. A condição de parada é quando um elemento não possui próximo, significando que ele é o último elemento da lista e deve retornar um. Se houver um próximo elemento, então a contagem deve retornar um mais a contagem do próximo.

Listagem 4.2 - Exemplo de elemento de lista ligada:

```
public class Elemento{
    private Object valor;
    private Elemento proximo;

    public Elemento(Object valor){
        this.valor = valor;
    }
    public Object getValor(){
        return valor;
    }
    public Elemento getProximo(){
        return proximo;
    }
    public void setProximo(Elemento proximo){
        this.proximo = proximo;
    }
    public int contar(){
        if(proximo == null)
            return 1;
        return 1 + proximo.contar();
    }
}
```

Até então não tem muita diferença entre as estruturas que podem ser construídas em uma linguagem estruturada, como C, em que podem-se criar estruturas de da-

dos utilizando a construção `struct`. O grande diferencial dessa definição recursiva de estruturas é a utilização do polimorfismo. Dessa forma, o tipo base utilizado na estrutura pode ser uma interface ou uma superclasse que possuem diversas implementações. Sendo assim, diversos tipos podem ser utilizados na criação da estrutura e a implementação de um pode ser abstraída pelos outros. Adicionalmente, essa estrutura também pode ser facilmente estendida através da criação de novos tipos que possuam a mesma implementação.

A Figura 4.1 apresenta diferentes possibilidades para a implementação da composição recursiva. Uma delas é ela ser definida na superclasse. Dessa forma, todas as subclasses poderão ser compostas por uma classe da hierarquia. Outra possibilidade é quando a composição recursiva é definida em uma ou mais subclasses. Nessa opção, apenas alguns tipos podem ser utilizados para a implementação da recursividade e outros serão pontos finais na estrutura. Nesse capítulo serão apresentados padrões que utilizam essas diferentes alternativas de implementação.

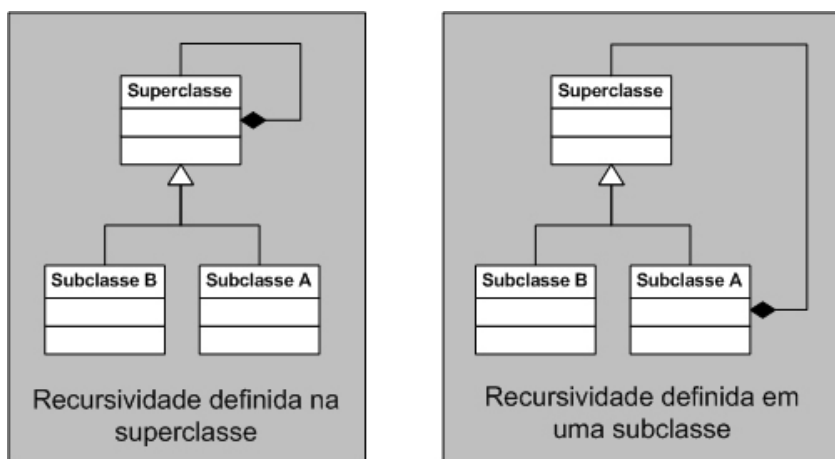



Figura 4.1: Possibilidades para Composição Recursiva

É importante ressaltar que existem variações das possibilidades apresentadas. A composição, por exemplo, pode ser de uma ou múltiplas instâncias, o que geraria soluções diferentes. Dependendo do domínio, pode também haver mais de um atributo que possua uma definição recursiva, mas que possua diferentes significados para a classe. Em uma árvore, por exemplo, uma lista de nós pode representar os nós filhos e um atributo simples pode representar o pai.

4.2 Composite - Quando um Conjunto é um Indivíduo

"Para sempre é composto de agoras."
– Emily Dickinson

Um dos erros de pessoas que estão iniciando na orientação a objetos é modelar o coletivo estendendo o indivíduo. Uma cesta de maçãs **não é uma** maçã para que o uso de herança seja justificado. Similarmente, um conjunto de processos **não é um** processo, logo também não é correto utilizar herança para representá-lo. Normalmente isso é feito para a subclasse invocar diretamente os métodos da superclasse e, dessa forma, a subclasse define um novo conjunto de métodos, ignorando a interface pública da superclasse.

Apesar do que foi dito, existem situações em que o conjunto também representa um único indivíduo. Um conjunto de produtos pode ser um produto? A resposta é sim dependendo do contexto. Uma loja ou um supermercado frequentemente montam kits com diversos produtos que são vendidos como um produto único. Como representar isso em um sistema? A questão é que esse conjunto de produtos terá uma lógica diferente para diversos fatores, como para o cálculo do preço, por exemplo. Porém, em outros cenários, ele deve ser considerado como um produto comum.

Essa seção apresenta o padrão **`Composite`** cujo objetivo é prover uma solução para objetos que representam um conjunto de objetos, mas que compartilham a mesma abstração deles. Ele tem o potencial de encapsular uma lógica complexa, dividindo-a em uma hierarquia de classes e simplificando a solução final. Todos que já utilizaram um **`Composite`** em um projeto real sabem que é um padrão realmente muito poderoso.

Apresentando o Padrão Composite

O padrão **`Composite`** possui o objetivo de permitir que a mesma abstração possa ser utilizada para uma instância simples e para seu conjunto. A Figura 4.2 apresenta a estrutura básica usada no padrão. Uma abstração apresenta uma interface básica com as operações que devem ser executadas tanto nos objetos simples quanto nos compostos. A classe `Simples` representa um único indivíduo e a classe `Composto` representa o conjunto. Observe que existe uma relação de composição entre `Composto` e `Abstracao`, que mostra que um objeto da classe `Composto` pode ser composto por outro também da classe `Composto`.

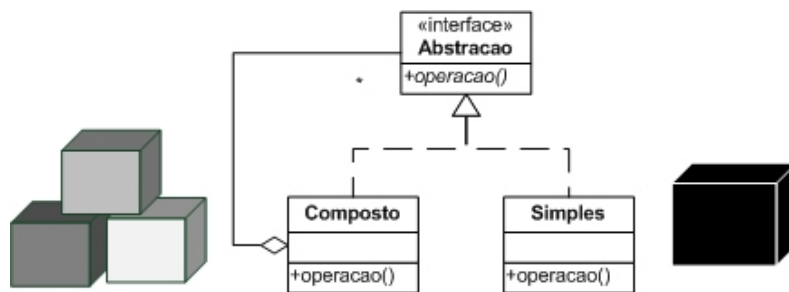



Figura 4.2: Estrutura Básica do Composite

As operações definidas na abstração devem ser implementadas em todas as subclasses. Para as subclasses simples, que não forem compostas por outras do tipo da abstração, pode ser uma implementação direta. Por exemplo, uma classe `Produto` pode disponibilizar um método para retornar o preço e possuir apenas o acesso a um atributo. O segredo do padrão **`Composite`** está na implementação dos métodos compostos, que devem invocar os métodos das instâncias que o compõe e executar uma lógica para criar um resultado único. No exemplo do produto, o produto composto, uma classe `KitProdutos`, para retornar o preço ele poderia somar o preço dos produtos que o compõe e dar um desconto em cima.

Os objetos organizados com o padrão **`Composite`** acabam seguindo uma estrutura de árvore. As folhas seriam as classes simples, que não possuem a composição. Apesar de no diagrama da Figura 4.2 apenas um tipo de classe simples ser representado, é importante ressaltar que podem haver diversas classes desse tipo com diferentes implementações. Os ramos seriam representados pelas classes compostas, que seriam compostas pelos seus filhos. O número de filhos de uma classe composta varia com a necessidade da aplicação, podendo ser duas instâncias ou um conjunto.

O grande benefício do **`Composite`** é tornar totalmente transparente para os clientes o fato de estarem trabalhando com um objeto simples ou composto. Isso torna muito simples adicionar uma implementação de um objeto composto em uma hierarquia já existente na aplicação.

> **COMPOSITE EM COMPONENTES GRÁFICOS**
>
> Um contexto em que o padrão **`Composite`** é muito utilizado é para a criação de componentes gráficos. Muitos são formados a partir da composição de outros componentes existentes, porém são tratados como um só. Por exemplo, um componente de seleção de arquivos normalmente possui botões e uma caixa de texto. Dessa forma, quando um método `setEnable()` for chamado, ele deverá coordenar a invocação desse método em todos os seus componentes internos.
>
> Em Java, esse conceito é utilizado em suítes gráficas para aplicações desktop, como no Swing e no SWT, e em frameworks com componentes para páginas web, como o JSF. O framework SwingBean, por exemplo, possui um que representa um formulário e é composto por diversos outros componentes.

Modelando a Representação de Trechos Aéreos

Para exemplificar a aplicação do padrão **`Composite`**, vamos modelar as classes que representam trechos aéreos em uma aplicação de turismo. Para esse exercício vamos considerar que as informações principais de um trecho aéreo são sua origem, seu destino e o seu preço. Muitas vezes, quando um cliente solicita um voo a partir de uma origem e um destino, não existe um voo direto que conecta aqueles dois aeroportos. Nesse caso, normalmente busca-se então um voo que pousa em um aeroporto intermediário, de onde o cliente pode pegar um outro voo até o seu destino. Nesse caso, além do preço dos dois trechos, deve-se adicionar a taxa de conexão cobrada pelo aeroporto intermediário. A questão é independente do trecho aéreo ser composto por mais de um trecho ou não, em certos pontos o sistema deve tratá-los de forma similar.

O primeiro passo é a definição de uma interface que irá definir quais serviços serão disponibilizados pela classe que representar o trecho simples e o composto. Pela descrição do problema, o trecho deve ser capaz de retornar a origem, o destino e o preço. Sendo assim, a listagem a seguir apresenta a interface `TrechoAereo` que define métodos para recuperação dessas informações. Em seguida, é apresentado o código da classe `TrechoSimples`, que possui uma implementação cuja lógica é apenas o acesso a atributos.

Listagem 4.3 - Interface que define os serviços de um trecho aéreo:

```
public interface TrechoAereo {
    public String getOrigem();
    public String getDestino();
    public double getPreco();
}
```

Listagem 4.4 - Representação de um trecho aéreo simples:

```
public class TrechoSimples implements TrechoAereo{
    private String origem;
    private String destino;
    private double preco;

    public TrechoSimples(String origem, String destino, double preco) {
        this.origem = origem;
        this.destino = destino;
        this.preco = preco;
    }
    public String getOrigem() {
        return origem;
    }
    public String getDestino() {
        return destino;
    }
    public double getPreco() {
        return preco;
    }
}
```

Para a implementação do trecho composto, será aplicado o padrão **`Composite`**. Nesse caso, o trecho composto possui dois atributos, respectivamente para representar o primeiro e o segundo trecho que o compõe. Também existe um atributo que representa a taxa de conexão a ser paga de forma adicional ao preço de cada trecho individualmente. Note que no construtor é feita uma verificação se o segundo trecho começa no mesmo local em que o primeiro termina. Pode ser observado que as operações que precisavam ser implementadas delegam parte da execução para os trechos que compõem o objeto.

Listagem 4.5 - Representação do trecho aéreo composto:

```
public class TrechoComposto implements TrechoAereo {
    private TrechoAereo primeiro;
    private TrechoAereo segundo;
    private double taxaconexao;

    public TrechoComposto(TrechoAereo primeiro, TrechoAereo segundo,
            double taxaconexao) {
        this.primeiro = primeiro;
        this.segundo = segundo;
        this.taxaconexao = taxaconexao;
        if(primeiro.getDestino() != segundo.getOrigem())
            throw new RuntimeException("O destino do primeiro" +
                    "não é igual a origem do segundo");
    }
    public String getOrigem() {
        return primeiro.getOrigem();
    }
    public String getDestino() {
        return segundo.getDestino();
    }
    public double getPreco() {
        return primeiro.getPreco() + segundo.getPreco() +
                taxaconexao;
    }
}
```

A Figura 4.3 apresenta um exemplo de como um voo com três trechos seria representado utilizando a estrutura de classes apresentada. A seguir também é mostrada uma listagem com o código para a criação dos trechos simples e compostos. Observe pelo exemplo, que `TrechoComposto` pode ser composto tanto por instâncias de `TrechoSimples` quanto por instâncias do próprio `TrechoComposto` de forma transparente, criando uma estrutura de árvore.

Figura 4.3: Representação de trechos aéreos

Listagem 4.6 - Criação de trechos compostos:

```
TrechoSimples ts1 = new TrechoSimples("São Paulo", "Brasília", 500);
TrechoSimples ts2 = new TrechoSimples("Brasília", "Recife", 300);
TrechoSimples ts3 = new TrechoSimples("Recife","Natal", 350);
TrechoComposto tc1 = new TrechoComposto(ts2, ts3, 30);
TrechoComposto tc2 = new TrechoComposto(ts1, tc1, 50);
```

Por que TrechoComposto não pode estender TrechoSimples?

Pode parecer tentador a princípio fazer com que `TrechoComposto` estenda `TrechoSimples` em vez de criar uma interface comum entre as duas. Se isso ocorresse, a classe `TrechoComposto` iria simplesmente ignorar a estrutura de dados da classe `TrechoSimples` e ainda teria que lidar com seu construtor. Isso geraria um mal cheiro de código chamado de *Refused Bequest*, que ocorre quando uma subclasse não utiliza a estrutura e os métodos herdados da sua superclasse. Para que aquela herança é dada para a classe se ela não a utiliza?

4.3 Encadeando Execuções com Chain of Responsibility

"Nenhuma corrente pode ser mais forte que seu elo mais fraco."
– Provérbio Popular

Quando em um software uma funcionalidade é executada, normalmente existem diversos passos que precisam ser executados em sequência. A dificuldade ocorre quando precisa-se de flexibilidade na configuração desses passos. Por exemplo, pode ser necessária a modificação da ordem dos passos e até mesmo para a inserção de novos passos. Um outro objetivo seria a possibilidade de reutilização desses passos em outros processamentos ou para outras aplicações.

`Chain of Responsibility` é um padrão de projeto que cria uma cadeia de execução na qual cada elemento processa as informações e em seguida delega a execução ao próximo da sequência. Em sua implementação tradicional, os elementos são percorridos até que um deles faça o tratamento da requisição, encerrando a execução depois disso. Como alternativa, também é possível criar uma cadeia de execução onde cada um executa sua funcionalidade até que a cadeia termine ou ela seja explicitamente finalizada por um dos elementos.

Recuperação de um Arquivo Remoto

Para ficar mais concreto o entendimento do padrão, imagine que um arquivo, como uma lista de certificados digitais revogados, é atualizado de forma periódica em um servidor remoto, contendo a data em que sua validade é expirada. Para tornar recuperações frequentes mais eficientes a aplicação criou dois tipos de cache, sendo um em memória e outro na base de dados. Dessa forma, primeiro verifica-se se existe um arquivo válido em memória, em seguida no banco de dados e somente então, se não for encontrado, o arquivo é recuperado do servidor remoto.

Em uma implementação tradicional, toda essa lógica seria implementada na mesma classe, com diversos condicionais em sequência para verificar se uma determinada estratégia seria adequada para recuperação do arquivo. Utilizando o padrão **`Chain of Responsibility`**, cada possibilidade é implementada em uma classe diferente, as quais são encadeadas em um fluxo de execução. Cada uma delas verifica se o arquivo existe e é válido, retornando o mesmo em caso positivo e delegando a execução para a próxima instância da cadeia em caso negativo. A Figura 4.4 representa essa cadeia de execução.

Figura 4.4: Cadeia de Execução para Recuperação de Arquivo

Para implementarmos essa sequência de execução, o primeiro passo é definirmos uma superclasse que define um elemento da cadeia. A classe `RecuperadorArquivo`, apresentada na próxima listagem, possui um atributo do seu mesmo tipo chamado `proximo` que representa o próximo elemento. Essa classe também define um método abstrato chamado `recuperarArquivo()` que é um *hook method* que deve ser implementado pelas subclasses para a recuperação do arquivo de acordo com a estratégia desejada. Além disso, o método `chamarProximo()` é responsável por verificar se existe um próximo elemento e invocá-lo. Adicionalmente, o método `recuperar()` tenta recuperar o arquivo, retornando-o se ele for válido e chamando o próximo elemento da cadeia em caso negativo.

Listagem 4.7 - Classe abstrata que define a lógica do Chain of Responsibility:

```
public abstract class RecuperadorArquivo {
    private RecuperadorArquivo proximo;

    public RecuperadorArquivo(RecuperadorArquivo proximo) {
        this.proximo = proximo;
    }
    public Arquivo recuperar(String nome){
        Arquivo a = recuperaArquivo(nome);
        if(a==null || !a.isValido())
            return chamarProximo(nome);
        else
            return a;
    }
```

```
    protected Arquivo chamarProximo(String nome) {
        if(proximo == null)
            throw new RuntimeException("Não foi possível " +
            " recuperar o arquivo");
        return proximo.recuperar(nome);
    }

    protected abstract Arquivo recuperaArquivo(String nome);
}
```

Cada subclasse de `RecuperadorArquivo` deve implementar o método `recuperaArquivo()` buscando o arquivo em uma fonte diferente. Pela descrição do problema, as implementações irão buscar o arquivo no banco de dados, de um cache em memória e de em um servidor remoto. A listagem a seguir apresenta a classe que faz a busca em um cache em memória para exemplificar como seria uma dessas implementações. O método `recuperaArquivo()` verifica se existe o arquivo no mapa e retorna nulo caso não exista. Para armazenar novos arquivos no mapa, o método `chamarProximo()` também é implementado para permitir que se tenha acesso ao arquivo retornado pelo próximo.

Listagem 4.8 - Classe que define o recuperador que define o cache em memória:

```
public class RecuperadorCacheMemoria extends RecuperadorArquivo {
    private Map<String, Arquivo> cache = new HashMap<>();

    public RecuperadorCacheMemoria(RecuperadorArquivo proximo) {
        super(proximo);
    }
    protected Arquivo recuperaArquivo(String nome) {
        if(cache.containsKey(nome))
            return cache.get(nome);
        return null;
    }
    protected Arquivo chamarProximo(String nome) {
        Arquivo a = super.chamarProximo(nome);
        cache.put(nome, a);
        return a;
    }
}
```

Alguém viu um Template Method?

Desafio o leitor a observar melhor o exemplo de código dessa seção e achar uma implementação do padrão **Template Method**! O método `recuperar()` da classe `RecuperadorArquivo` implementa uma lógica comum aos elementos da cadeia de execução e delega os pontos variáveis a *hook methods* que serão implementados pelas subclasses. Observe que o método `chamarProximo()`, também chamado pelo **Template Method**, apesar de nao ser abstrato, também pode ser considerado um *hook method*. Pelo exemplo da classe `RecuperaCacheMemoria` pode-se perceber como ele pode ser sobrescrito pela subclasse para a adição da funcionalidade.

Como já ressaltamos, é comum haver uma ligação forte entre diferentes padrões de projeto. Muitas vezes eles aparecem em conjunto e até é difícil diferenciá-los.

Estrutura e Alternativas de Implementação

O padrão **Chain of Responsibility** coloca cada elemento que irá participar do processamento da lógica como o elemento de uma lista ligada, onde cada um tem acesso ao próximo na cadeia de execução. A Figura 4.5 mostra como a composição recursiva é utilizada na implementação da estrutura de classes. Nesse caso, uma *hook class* do mesmo tipo é definida na superclasse, significando que todo tipo de elemento da cadeia pode ser composto por um próximo. Caso ele seja nulo, ou possua um elemento que represente um **Null Object**, será marcado o final da cadeia de execução.

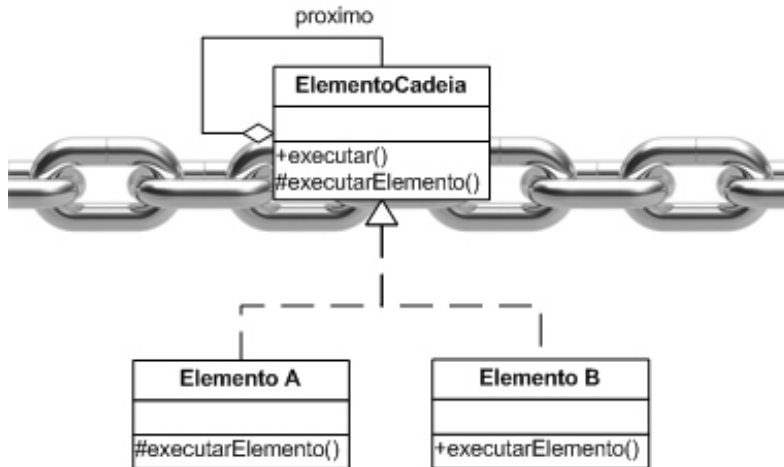



Figura 4.5: Estrutura do Chain of Responsibility

Nessa representação, o método `executar()` disponibiliza como API pública a execução de todos elementos da cadeia e o método `executarElemento()` representa a execução da lógica somente daquele elemento. Dessa forma, o método `executar()` invoca primeiro a lógica implementada na própria classe e em seguida invoca de forma recursiva o próprio método `executar()` no próximo elemento.

Como alternativa de implementação, o método `executarElemento()` pode receber como parâmetro o próximo elemento, sendo também responsável pela sua invocação. Dessa forma, cabe à implementação de cada classe decidir em que situações o próximo deve ser invocado e em que momento isso deve acontecer. Utilizando essa alternativa, cada implementação pode incluir lógica que será executada tanto antes quanto depois da execução dele.

A partir da implementação do **`Chain of Responsibility`** adquire-se uma grande flexibilidade para a configuração da lógica que será executada. Por exemplo, facilmente um novo elemento da cadeia pode ser introduzido, ou um existente pode ser excluído. A própria ordem em que os elementos são organizados pode gerar uma grande diferença no final. Por exemplo, para a utilização do `RecuperadorArquivo` em dispositivos com memória RAM limitada, o elemento que faz o cache em memória poderia facilmente ser excluído da cadeia. Em um outro cenário em que o arquivo pudesse estar disponível em mais de um servidor, mais instâncias do elemento que o busca remotamente com endereços diferentes poderiam ser incluídas.

Filtros em Aplicações Web

Um exemplo do uso do **Chain of Responsibility** em APIs do Java EE são os filtros de aplicações web [21]. Um exemplo da estrutura básica de um filtro está representada na próxima listagem. A classe que representa o filtro deve implementar a interface `Filter` e, além de implementar os métodos `init()` e `destroy()` que servem respectivamente para a inicialização e finalização, ainda é preciso incluir o método `doFilter()` que é onde a lógica será efetivamente implementada.

Listagem 4.9 - Exemplo de Filtro para Aplicações Web:

```
public class FiltroExemplo implements Filter{

  public void doFilter(ServletRequest req,
                       ServletResponse resp,
                       FilterChain chain){

    //antes do próximo filtro
    chain.doFilter(req,resp);
    //depois do próximo filtro

  }
  public void destroy{}
  public void init(FilterConfig config){}
}
```

O método `doFilter()` recebe como um dos parâmetros uma instância da classe `FilterChain`, que representa justamente a cadeia de filtros que está sendo executada. Ao ser chamado o método `doFilter()` nessa instância, o controle da execução é passado para o próximo filtro ou, se não houver próximo, para o servlet que irá tratar a requisição. Observe que como o próximo filtro é invocado dentro do método `doFilter()`, pode ser adicionada funcionalidade tanto antes quanto depois da sua execução. Inclusive o próximo filtro nem precisa ser invocado, como no caso da requisição ser direcionada a uma página de erro. Outra coisa que pode ser feita é a modificação dos parâmetros que serão passados para os próximos filtros, inclusive encapsulando-os com a utilização de **Proxies** (um padrão que será visto no próximo capítulo).

Um dos usos mais clássicos para filtros em aplicações web é para verificar se o usuário está logado. Nesse caso, ele verifica se existe um usuário válido na sessão e

redireciona para a página de login em caso negativo. Porém existem vários outros usos para filtros, como: realização de controle de acesso; fazer uma busca por parâmetros maliciosos; iniciar e finalizar uma transação com a base de dados; e modificar a saída gerada pelo servlet. Em geral os filtros são utilizados para funcionalidades que precisam ser executadas em diversas requisições diferentes. A utilização do **`Chain of Responsibility`** em sua implementação permite que a ordem dos filtros seja modificada e que filtros possam ser facilmente introduzidos ou removidos.

Essa modelagem utilizada para os filtros é mesma dos *interceptors* do framework Struts 2 [7]. Um *interceptor* tem uma função bem parecida, que é de realizar algum processamento antes e/ou depois do tratamento de uma requisição web. Grande parte das funcionalidades do framework, como a injeção dos parâmetros do *request* ou a introdução de objetos da sessão do usuário, é implementada através dos *interceptors*. Os interceptors que atuam para cada requisição são configurados através de um arquivo XML. Nesse caso, a utilização do padrão **Chain of Responsibility** permite que esses componentes do framework possam ser facilmente removidos ou que novos componentes específicos da aplicação possam ser adicionados.

4.4 Refatorando para Permitir a Execução de Múltiplas Classes

Uma evolução comum que ocorre em aplicações é a necessidade de que mais uma ação seja realizada como resposta a um evento. Por exemplo, imagine que ao receber uma informação a aplicação precise persisti-la em uma base de dados. Com o tempo, novos requisitos vão surgindo e exigem que novas ações sejam tomadas naquele cenário, como a chamada de um serviço remoto ou o registro em uma trilha de auditoria. O objetivo dessa refatoração é permitir que, na chamada de um método, diversas classes possam ser executadas. Isso deve ser realizado de forma que não haja um acoplamento entre essas classes e que isso seja transparente aos que invocam.

Para que essa refatoração possa ser realizada, é importante que a execução dessa funcionalidade já esteja isolada utilizando a composição simples. Ou seja, que a partir de uma *hook class* configurada seja possível trocar a lógica que será executada, mas não incluir múltiplas classes para a execução. Diferentemente do padrão **`Observer`**, no qual a classe observada precisa gerenciar a existência de múltiplos observadores, nesse caso é desejável que a existência de múltiplas execuções seja transparente para a classe.

Para mostrar a refatoração para os dois padrões apresentados nesse capítulo,

será retomado o exemplo da classe `GeradorArquivo`. O objetivo será permitir que diversos pós-processadores possam atuar em cima do arquivo gerado de forma transparente para a classe principal. Após o padrão **Bridge**, existe uma interface chamada `PosProcessador` que deve ser implementada por qualquer implementação que irá compor a classe `GeradorArquivos`. No exemplo, os dois pós-processadores existentes respectivamente criptografam e compactam o arquivo após a sua geração. Observe que se ambos forem utilizados, a ordem de processamento irá alterar o resultado final, pois um arquivo criptografado e compactado é diferente de um arquivo compactado e criptografado.

As seções a seguir descrevem como cada um dos padrões apresentados nessa seção poderiam ser implementados para resolver o problema com os requisitos descritos.

Introduzindo um Composite

Para implementar o padrão **Composite**, o que precisa ser feito é a implementação de uma classe que possua a mesma interface de um pós-processador e seja capaz de coordenar a execução de diversos pós-processadores. A listagem a seguir mostra a classe `PosProcessadorComposto` que desempenha esse papel. Observe que ela é composta por um array de pós-processadores que são recebidos no construtor. Adicionalmente, o método `processar()` executa o mesmo método de forma recursiva nas instâncias de `PosProcessador` que compõem a classe.

Listagem 4.10 - Criação de um pós-processador composto:

```
public class PosProcessadorComposto implements PosProcessador{

    private PosProcessador[] processadores;

    public PosProcessadorComposto(PosProcessador... p){
        processadores = p;
    }
    public byte[] processar(byte[] bytes){
        for(PosProcessador p : processadores){
            bytes = p.processar();
        }
        return bytes;
    }
}
```

A grande vantagem do uso do **Composite** nesse caso é que ele não exige

nenhuma mudança nas classes existentes. A classe `PosProcessadorComposto` pode ser adicionada facilmente na classe `GeradorArquivo` e todos os outros pós-processadores podem facilmente ser incluídos nela. A dificuldade de usar o **Composite** irá aparecer em cenários onde a integração entre as classes envolvidas não acontece sempre da mesma forma. Isso pode exigir a criação de diversas classes que desempenham esse papel, mas possuem uma lógica de integração diferente.

Refatorando para Chain of Responsibility

Uma outra abordagem para a refatoração seria implementar o padrão **Chain of Responsibility** para permitir que os pós-processadores possam ser organizados em uma cadeia de processamento. Isso vai exigir uma mudança na abstração que representa um pós-processador para incluir a composição recursiva, ou seja, para a adição de uma *hook class* que represente o próximo processador da cadeia. A interface `PosProcessador` precisaria ser representada como uma classe abstrata, cuja listagem está apresentada a seguir.

Listagem 4.11 - Classe PosProcessador para a criação de uma cadeia de processamento:

```
public abstract class PosProcessador{

    private PosProcessador proximo;

    public PosProcessador(PosProcessador prox){
        proximo = prox;
    }
    public PosProcessador(){
        proximo = nem PosProcessadorNulo();
    }
    public byte[] processarCadeia(byte[] bytes){
        bytes = processar(bytes);
        return proximo.processarCadeia(bytes);
    }
    protected abstract byte[] processar(byte[] bytes);
}
```

A classe `PosProcessador` possui o atributo `proximo` de seu próprio tipo, que deve ser passado no construtor. Caso seja o final da cadeia, um construtor vazio pode ser invocado, o qual irá configurar um **Null Object** no atributo

`proximo`. O método abstrato `processar()` representa a lógica de processamento específica daquela implementação que será invocada durante a execução da cadeia - e, por ele não poder ser invocado de fora, sua visibilidade é `protected`. O método `processarCadeia()` é o responsável pela execução da cadeia e invoca não somente o método `processar()` da própria classe, quanto o método `processarCadeia()` do atributo `proximo`.

Diferentemente da implementação do **Composite**, essa refatoração tem um pequeno impacto nas outras classes envolvidas. Por exemplo, as classes que invocam pós-processadores precisarão alterar o método que está sendo chamado de `processar()` para `processarCadeia()`. Ele poderia ser mantido, porém às custas de uma mudança no nome do *hook method* que precisa ser criado nos pós-processadores. Ou seja, como o método disponível publicamente não seria mais o ponto de extensão, o nome de algum deles precisa ser modificado. Outras mudanças devem ser feitas nas subclasses, que precisam disponibilizar o novo construtor que recebe um `PosProcessador` como parâmetro e alterar de `implements` para `extends` a relação com `PosProcessador`, que não é mais uma interface.

4.5 Considerações Finais do Capítulo

Esse capítulo apresentou conceitos a respeito da composição recursiva, na qual objetos são compostos por outros que possuem uma de suas abstrações. Com a utilização do polimorfismo, é possível combinar diversas implementações de uma mesma abstração formando uma estrutura flexível e complexa. A partir da criação de novas classes e sua inclusão em um de seus pontos pode-se estender o seu comportamento. A própria organização das classes não deixa de ser um ponto que pode ser adaptado e configurado.

O padrão **Composite** utiliza esse conceito para permitir a representação de objetos compostos, em outras palavras, objetos que combinam a funcionalidade de outros do mesmo tipo. Com esse padrão, uma das subclasses é quem possui a composição recursiva. Nesse padrão as classes se organizam em forma de árvore, sendo as compostas os nós internos e as outras, as folhas.

Já o **Chain of Responsibility** cria uma estrutura parecida com uma lista ligada, na qual cada elemento possui uma referência para o próximo. A composição recursiva normalmente é implementada na superclasse, pois todo tipo de elemento deve poder ter um próximo. A partir desse encadeamento de classes, é possível configurar quais elementos devem ser incluídos no processamento.

A utilização desse tipo de estrutura normalmente é ligada à necessidade de flexibilidade e configurabilidade da lógica da aplicação. A recursividade da solução permite o desacoplamento entre os elementos e a criação de diversas possibilidades de configuração.

Capítulo 5

Envolvendo Objetos

"Não existe colher. Você verá que não é a colher que se dobra, apenas você mesmo. "
– Garoto com a colher, Matrix

O encapsulamento é uma característica que faz com que os clientes das classes precisem conhecer apenas a sua interface, abstraindo sua implementação interna. O polimorfismo permite que a instância de uma classe possa ser atribuída a uma variável, desde que obedeça à abstração utilizada como tipo. Essas duas características fazem com que exista uma ignorância, no sentido correto da palavra, da classe cliente em relação à classe com que ela realmente está lidando.

A partir desses conceitos, é possível que uma classe envolva uma instância sem que a cliente saiba que está lidando com outro objeto. Dessa forma, ela servirá como intermediária entre a classe cliente e a classe alvo. Com isso, ela obtém o controle da execução antes e depois de cada invocação de método. Isso permite que, por exemplo, funcionalidades sejam adicionadas e que validações sejam realizadas. Pode-se inclusive interceptar a execução de um método, retornando um valor ou lançando um erro mesmo sem invocar o método original.

Esse capítulo apresenta os padrões e conceitos relacionados a esse encapsulamento de objetos. Este mesmo conceito pode ser utilizado para proteger um objeto, adicionar funcionalidades e até mesmo adaptar interfaces. Será mostrado que um dos grandes poderes dessa técnica é fazer o código cliente pensar que ainda está lidando com o objeto original e fazer que esse objeto não saiba que está sendo envolvido.

5.1 Proxies e Decorators

"O olhar é, antes de mais nada, um intermediário que remete de mim a mim mesmo."
– Sartre

Diferentemente dos capítulos anteriores, vou começar já apresentando dois padrões ao mesmo tempo. O motivo para isso é que o padrão **`Proxy`** possui uma estrutura muito parecida com o padrão **`Decorator`**, sendo que a diferença é basicamente a motivação e o contexto no qual os padrões são aplicados.

A ideia básica desses padrões é criar uma classe que envolve uma outra do mesmo tipo. Dessa forma, ela pode ser passada de forma transparente como se fosse a classe original para quem a irá utilizar. A Figura 5.1 representa o funcionamento desses padrões.

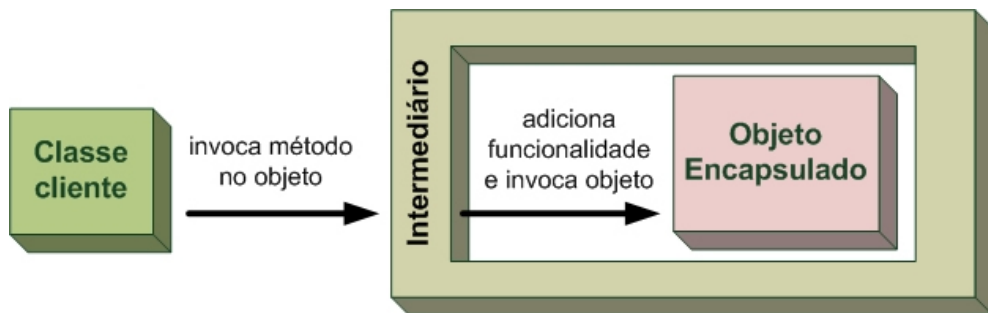



Figura 5.1: Envolvendo um objeto

A utilização desse tipo de técnica traz uma nova perspectiva para a modelagem de classes, pois ao invés de concentrar todos os aspectos de um método em apenas uma, é possível tratar diferentes questões em diferentes camadas. Assim a funcionalidade da classe que está encapsulando pode ser utilizada para várias implementações.

Para ficar mais concreto, uma validação de parâmetros que normalmente é feita dentro dos métodos poderia ser colocada em uma classe que envolve a classe principal. Dessa forma, essa validação poderia ser reutilizada para diversas implementações.

A Estrutura dos Padrões

A estrutura do **`Proxy`** e do **`Decorator`** utiliza a composição recursiva. Isso significa que ambos são compostos por uma classe que possui a mesma abstração que eles. A estrutura desses padrões está apresentada na Figura 5.2. Observe que são muito similares e permitem que a instância de uma classe encapsule um objeto e possa assumir o seu lugar. Dessa forma, os clientes dela não terão conhecimento de se estão lidando com a original ou com uma que a está encapsulando.

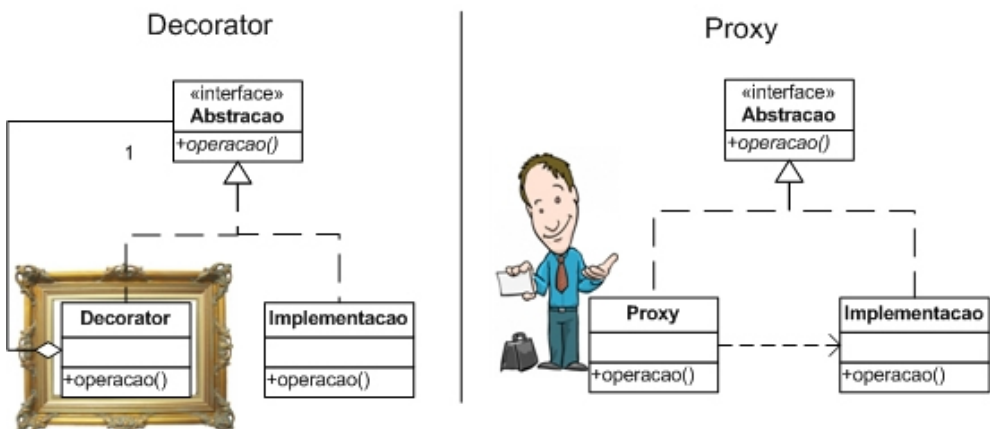



Figura 5.2: Estrutura dos padrões Proxy e Decorator

O padrão **`Decorator`** recebeu esse nome relacionado ao fato de "decorar" uma classe existente adicionando uma nova funcionalidade. Imagine como se a classe principal fosse um quadro e o **`Decorator`** fosse a moldura, que está acrescentando elementos no visual do quadro. O principal objetivo desse padrão é acrescentar funcionalidades a classes existentes de uma forma transparente a quem as utiliza. Isso também pode ser utilizado para uma melhor distribuição de responsabilidades, permitindo que a principal se foque na regra de negócio central e que outras classes que a encapsulam cuidem de outras funcionalidades.

O padrão **`Proxy`** está mais ligado a prover um objeto para servir como intermediário na comunicação com um outro principal. Um exemplo comum de utilização

desse padrão é para representar localmente objetos que estão em servidores remotos. Dessa forma, o proxy encapsula a lógica de acesso remoto fazendo parecer que o objeto está sendo acessado localmente. Um outro exemplo é para a criação de objetos "caros", ou seja, que possuem um alto custo computacional de consumo de recursos para criação. O **`Proxy`** pode servir como uma representação para esse objeto, permitindo que o mesmo seja criado somente quando for necessário.

Uma coisa interessante da estrutura de ambos os padrões é sua composição recursiva que permite que seja feito um encadeamento de classes, de forma similar ao **`Chain of Responsability`**. Sendo assim, um **`Proxy`** ou **`Decorator`** pode não somente encapsular a classe original, mas também encapsular uma classe que já esteja encapsulada com um outro **`Proxy`** ou **`Decorator`**. Dessa forma, a classe fica com diversas camadas, cada uma com uma responsabilidade diferente, encapsulando a regra de negócio principal.

A principal diferença entre os dois padrões está no objetivo de cada um. Enquanto o **`Decorator`** visa adicionar novas funcionalidades, o **`Proxy`** serve mais como uma proteção ao objeto principal. Em termos de estrutura, o **`Decorator`** costuma permitir que o objeto encapsulado seja passado no construtor, ou configurado de alguma outra forma. Desse modo, a composição é feita utilizando a abstração, permitindo que qualquer implementação seja encapsulada. Já o **`Proxy`**, muitas vezes, protege um objeto específico e a criação do objeto encapsulado ocorre dentro dele. No caso de um **`Proxy`** que faça acesso remoto, ele nem compõe a classe propriamente dita, mas utiliza as classes de acesso à rede para delegar as chamadas.

Porém essa diferença estrutural é muito sutil e pode até mesmo ser considerada como um detalhe de implementação. Em relação aos objetivos, a proteção ao acesso feita pelo **`Proxy`** não deixa de ser uma funcionalidade como no **`Decorator`**. Por esse motivo, daqui para frente nesse livro o padrão **`Proxy`** será citado referenciando ambos os padrões.

Para que eu preciso de uma interface se nunca vou mudar essa implementação?

Muitas vezes vejo desenvolvedores pensando duas vezes antes de criar uma abstração, no formato de uma interface, de algum componente importante do sistema. Por exemplo, vamos imaginar uma classe que retorne propriedades de um arquivo de configuração do sistema. Por mais que não exista a possibilidade de se criar uma nova implementação para essa classe, pode ser interessante que ela possua uma abstração.

Um dos motivos para isso é a possibilidade de se criar um `Proxy` que irá envolver essa implementação. Nesse exemplo da leitura do arquivo, poderia haver `Proxies` para funcionalidades como o armazenamento dos valores em memória e para a verificação da consistência desses valores. Sendo assim, antes de pensar que só vai haver uma classe, pense que a abstração também pode ser utilizada na criação de `Proxies` que podem encapsular a classe principal.

Se você por acaso não criou uma interface para uma classe que precisa de um `Proxy`, lembre-se que não é difícil refatorá-la com as refatorações automatizadas disponíveis nos IDEs. No Eclipse, por exemplo, ao extrair a interface, existe a opção para utilizá-la ao invés do tipo da classe onde for possível. Sendo assim, mais importante que criar a interface é obedecer ao encapsulamento, evitando depender de detalhes internos da classe.

Cenários para Aplicação de Proxies

A utilização de `Proxies` é um recurso tão poderoso que me arriscaria a dizer que é o padrão que mais utilizo. Antes de entrar em exemplos mais detalhados, acho importante mostrar alguns cenários onde a utilização dessa técnica pode ser feita. Acredito que isso ajuda a enxergar o potencial desses padrões.

Muitas vezes é necessária a criação de código de proteção para evitar que seja feita uma invocação de método com parâmetros inadequados ou em um contexto errado. Exemplos dessas situações seriam parâmetros que precisam estar em um formato especifico ou métodos que só podem ser invocados em determinados estados da classe. Outro cenário onde é importante proteger a classe é para requisitos

de segurança, evitando que uma funcionalidade seja acessada por um usuário não-autorizado ou bloqueando parâmetros maliciosos que tentam executar ataques de injeção (ver quadro). Para todos esses casos, para evitar que esse código de proteção fique misturado com o código do negócio, muitas vezes é interessante criar um proxy exclusivamente para bloquear esses acessos indevidos.

Ataques de Injeção

Um ataque de injeção é aquele que se aproveita de aplicações que se utilizam da concatenação de strings para a formação de um comando que será executado. Dessa forma, o atacante procura enviar um parâmetro malicioso para a aplicação visando a formação de um comando diferente durante a concatenação. Um dos exemplos mais famosos é o SQL Injection, no qual procura-se injetar partes de comandos SQL em uma string que formará uma consulta que será executada no banco de dados. Outro exemplo de ataque é o *Cross-site Scripting*, que faz o uso de injeção de javascript em parâmetros que são utilizados para a construção de páginas web.

Um outro cenário onde o uso de `Proxies` é bastante comum é para fazer cache da execução de métodos. Quando uma funcionalidade demorada precisa ser executada de forma repetida em uma aplicação, é interessante armazenar o resultado obtido na primeira execução e reutilizá-lo em novas invocações. Colocar isso junto com a própria funcionalidade pode tornar o código confuso, misturando duas responsabilidades no mesmo local. Dessa forma, um `Proxy` pode ser utilizado para encapsular a classe, armazenando o resultado da primeira execução e retornando-o sem executar a classe novamente nas próximas invocações.

Outra utilização clássica desse tipo de técnica é para tornar transparente o acesso a um sistema remoto. A um certo tempo atrás, a comunicação entre aplicações remotas precisava ser implementada através do acesso a bibliotecas de rede, sendo uma atividade difícil e trabalhosa. Hoje, grande parte das tecnologias para comunicação remota, como RMI, EJB ou mesmo classes clientes de webservices, encapsula todo esse processo dentro de um `Proxy`, o qual é responsável por acessar remotamente um componente disponibilizado pelo servidor. Esse `Proxy` implementa a mesma interface do objeto original, permitindo uma transparência desse processo para a aplicação, que pode acessar de forma indiferente o objeto local ou o remoto.

Em geral, esses padrões são utilizados para funcionalidades que são ortogonais a estrutura do software. Isso significa que elas cortam a estrutura do software estando presentes em diversas classes, não estando ligadas diretamente à sua funcionalidade. Por exemplo, uma funcionalidade de registro de auditoria ou de controle de acesso costuma ser necessária em diversas classes de um mesmo tipo. A utilização de um **Proxy** permitiria, não somente a reutilização disso em diversas implementações de uma abstração, como também permitiria que essa característica fosse modularizada, evitando seu espalhamento em diversas classes.

Proxies na API do Java

As APIs da linguagem Java estão repletas de exemplos de implementação dos padrões **Proxy** e **Decorator**. Como todo bom encapsulamento, se você já utilizou algum deles, provavelmente não percebeu a sua presença. Essa seção traz alguns exemplos de uso desses padrões, que podem servir como uma referência para o seu uso.

O primeiro exemplo está na API de coleções. Essa API possui algumas classes que servem para "decorar" as coleções existentes adicionando funcionalidades a elas. A classe `Collections` possui métodos estáticos como `synchronizedList(List<T> list)` que retorna a lista recebida com sincronização e `unmodifiableList(List<? extends T> list)` para criar uma lista não-modificável. O que na verdade esses métodos fazem é encapsular a lista passada em um **Proxy** que adiciona essa característica ao objeto retornado.

Outro uso importante do padrão **Proxy** acontece no RMI (Remote Method Invocation), que é a tecnologia utilizada em Java para a invocação remota de métodos. Nesse modelo, após a criação da classe que disponibilizará seus métodos remotamente, são gerados os chamados *stubs* e *skeletons*. Os *skeletons* são classes que recebem no servidor as requisições e invocam a classe remota. Já os *stubs* são **Proxies** que ficam no cliente e representam a classe remota, possuindo a mesma interface que ela.

A API padrão para o mapeamento objeto-relacional no Java EE, o JPA (Java Persitence API), possui uma funcionalidade muito interessante que permite um carregamento preguiçoso de listas associadas à classe principal. Isso significa que se eu tenho uma classe `Pessoa` que possui uma lista de instâncias de `Telefone` que são persistentes em uma base relacional, é possível que ao carregar uma `Pessoa`, sua lista de `Telefone` seja carregada somente na primeira vez em que ela é acessada. Isso é feito a partir de um **Proxy** que acessa a base de dados e popula a lista somente

em seu primeiro acesso.

Proxies Dinâmicos

Quando criamos uma classe para ser um **`Proxy`**, ela precisa implementar a mesma abstração da classe que está encapsulando para que possa ser passada no lugar dela. Em Java é possível criar os chamados **Proxies Dinâmicos**. Eles têm a capacidade de assumir dinamicamente uma interface, permitindo que seja utilizado para diversas interfaces. Esse é um recurso avançado da API de reflexão que foge ao escopo desse livro, porém é importante citar para que o leitor saiba da existência desse tipo de recurso.

5.2 Exemplos de Proxies

"Falar é fácil. Mostre-me o código."
– Linus Torvalds

Nesse capítulo, um longo caminho foi percorrido sem nem um exemplo de código. Para não decepcionar os leitores, introduzo esta seção dedicada a apresentar alguns exemplos de **`Proxy`**. A ideia é mostrar exemplos realistas que possam ser adaptados para outras aplicações. Cada subseção irá apresentar um contexto, a interface utilizada pelas classes e em seguida o código do **`Proxy`**.

Invocações Assíncronas

Muitas vezes em alguns sistemas temos funcionalidades que demoram a ser executadas. Nesses casos, é ruim deixar o usuário esperando, sendo que em alguns casos ele pode inclusive achar que o sistema travou. O que costuma acontecer é essas funcionalidades serem executadas em uma thread diferente. Abaixo segue a listagem da interface que abstrai os tipos de transação do sistema. Para as implementações, as informações da transação são armazenadas em atributos e normalmente passadas no construtor. Para mais detalhes sobre essa construção, ver o Capítulo 8 sobre o padrão **`Command`**.

Listagem 5.1 - Interface que define os serviços de um trecho aéreo:

```
public interface Transacao {
```

```
    public void executar();
}
```

Talvez o primeiro impulso fosse adicionar a criação de novas threads nas próprias implementações de `Transacao`. Porém isso iria gerar uma duplicação de código entre as classes que precisariam disso. Isso também causaria problemas se uma mesma transação precisasse ser executada na mesma thread ou em uma thread diferente dependendo do contexto. Nesse caso, a criação da thread na chamada da transação poderia resolver esse problema, mas não resolveria o problema da duplicação.

Uma forma de abordar isso seria a criação de um **`Proxy`** que encapsulasse uma transação e a invocasse de forma assíncrona. A classe `ProxyAssincrono` apresentada a seguir implementa essa funcionalidade. Ele pode ser utilizado em qualquer implementação de `Transacao`, evitando a duplicação, e inclusive a mesma classe pode ser utilizada com ou sem o **`Proxy`**.

Listagem 5.2 - Interface que define os serviços de um trecho aéreo:

```
public class ProxyAssincrono implements Transacao{

    private Transacao t;

    public ProxyAssincrono(Transacao t) {
        this.t = t;
    }
    public void executar() {
        Runnable r = new Runnable(){
            public void run() {
                t.executar();
            }
        };
        Thread t = new Thread(r);
        t.start();
    }
}
```

Realizando o Cache de um DAO

`Data Access Object`, também conhecido como **`DAO`**, é um padrão com o objetivo de isolar o acesso a dados de uma aplicação [6]. A partir dele, define-se uma interface para o acesso a dados e uma implementação para encapsular todas

as responsabilidades referentes à interface com um banco de dados. Apesar de ser um padrão fora do escopo desse livro, ele descreve uma prática bastante popular em aplicações que lidam com dados.

Esse exemplo se contextualiza em um software que possui um **DAO** que realiza o acesso a um banco relacional. Imagine que a interface dessa camada seja a apresentada na listagem a seguir. Normalmente um **DAO** costuma possuir mais métodos, porém nesse exemplo vamos considerar apenas três, respectivamente para recuperação, exclusão e gravação na base de dados. Imagine que a aplicação esteja tendo problemas de desempenho e tentando diminuir a quantidade de acessos a banco. Uma forma de fazer isso é guardando alguns objetos em memória e reaproveitá-los em leituras subsequentes. Como fazer isso com o menor impacto possível?

Listagem 5.3 - Interface que define os serviços de um trecho aéreo:

```
public interface DAO {
    public Identificavel recuperar(int id);
    public void excluir(int id);
    public void salvar(Identificavel obj);
}
```

Para não afetar nem os clientes nem os próprios **DAOs**, a solução seria colocar um **Proxy** encapsulando o **DAO** original para adicionar a funcionalidade de cache. A seguir é apresentada a implementação da classe `CacheDAO`. Ele possui um mapa que é utilizado para armazenar os objetos persistidos a partir do seu identificador. Os métodos `salvar()` e `excluir()` respectivamente adicionam e removem os objetos do mapa, mas apenas após delegar a chamada à classe que está encapsulando.

Listagem 5.4 - Interface que define os serviços de um trecho aéreo:

```
public class CacheDAO implements DAO {

    private DAO dao;
    private Map<Integer,Identificavel> cache;

    public CacheDAO(DAO dao) {
        this.dao = dao;
        this.cache = new HashMap<>();
    }
    public Identificavel recuperar(int id) {
```

```
        if(cache.containsKey(id))
            return cache.get(id);
        Identificavel obj = dao.recuperar(id);
        cache.put(id, obj);
        return obj;
    }
    public void excluir(int id) {
        dao.excluir(id);
        cache.remove(id);
    }
    public void salvar(Identificavel obj) {
        dao.salvar(obj);
        cache.put(obj.getId(), obj);
    }
}
```

Dessa forma, o método `recuperar()` primeiro verifica se o objeto está no cache e em caso positivo o retorna sem invocar o **DAO** encapsulado. Caso o objeto não esteja no cache, o **DAO** é então invocado e o objeto encapsulado é armazenado no cache antes de ser retornado. Observe que nesse exemplo o **Proxy** pode, em alguns casos, retornar para o cliente sem nem invocar o método do objeto encapsulado.

Vale ressaltar que o exemplo de cache apresentando é bem simples e não está preparado para lidar com uma série de questões importantes. Para exemplificar podem ser citadas a sincronização para diversos acessos paralelos, a expiração devido à possibilidade de atualização dos dados por outras origens e a distribuição no caso de aplicações em que os dados são acessados de várias máquinas. Caso em sua aplicação seja necessário um cache mais elaborado, sugere-se a utilização de padrões destinados à criação de cache para acesso a dados [19]. Porém, quanto mais complexo ele precisar ser, mais adequado é a utilização de um **Proxy** para isolar essa complexidade do resto da aplicação. Frameworks como o Hibernate já trabalham dessa forma e até possibilitam a escolha da sua biblioteca de cache, como o EhCache e o Infinispan.

Tentativas Repetidas de Acesso

Outra situação que não é incomum de acontecer é ser necessário adicionar uma funcionalidade em uma classe a qual não podemos modificar. Isso normalmente acontece quando essa classe faz parte de um framework ou biblioteca de classes que está sendo utilizada. Considere a interface representada a seguir como a

implementada por uma classe disponibilizada por uma empresa para buscar um arquivo em seus servidores remotos. Porém, devido a instabilidade da rede, existem muitas falhas ao tentar recuperar o arquivo.

Listagem 5.5 - Interface que define os serviços de um trecho aéreo:

```
public interface AcessoRemoto {
    public byte[] buscarArquivo(String url) throws IOException;
}
```

Para diminuir o número de falhas, uma alternativa é realizar tentativas repetidas de acesso ao arquivo. Como a classe não pode ser modificada e não é desejável modificar as classes que buscam o arquivo, uma alternativa é a criação de um **`Proxy`** que tente buscar o arquivo novamente no caso de uma falha. Na listagem a seguir é apresentada a classe `TentativasRepetidas` que encapsula uma instância do tipo `AcessoRemoto`. O método `buscarArquivo()` invoca o mesmo método no objeto que está sendo encapsulado, porém dentro de um bloco `try/catch` onde um possível erro é capturado incrementando o número de tentativas. Sendo assim, o método é invocado novamente até que nenhum erro seja retornado ou que o número limite de tentativas seja atingido.

Listagem 5.6 - Interface que define os serviços de um trecho aéreo:

```
public class TentativasRepetidas implements AcessoRemoto{

    private AcessoRemoto ar;
    private int maximoTentativas;

    public TentativasRepetidas(AcessoRemoto ar, int maximoTentativas) {
        this.ar = ar;
        this.maximoTentativas = maximoTentativas;
    }
    public byte[] buscarArquivo(String url) throws IOException{
        int tentativas = 0;
        IOException ultimoErro = null;
        while(tentativas < maximoTentativas){
            try{
                return ar.buscarArquivo(url);
            }catch(IOException ex){
                tentativas++;
                ultimoErro = ex;
```

```
            }
        }
        throw ultimoErro;
    }
}
```

5.3 Extraindo um Proxy

Normalmente os padrões **Proxy** e **Decorator** são aplicados quando uma funcionalidade precisa ser adicionada e deseja-se causar um impacto mínimo no código que já foi implementado. Nesse caso não é necessário refatorar a aplicação para a implementação do padrão. A necessidade de refatoração na direção de um **Proxy** pode ocorrer quando uma classe está inchada com muitas funcionalidades e deseja-se dividir essas responsabilidades. A refatoração para esse cenário seria extrair um **Proxy** com uma funcionalidade não ligada diretamente ao objetivo principal da classe. Esse procedimento poderia ser realizado diversas vezes para diferentes funcionalidades.

Um outro caso no qual essa refatoração seria necessária é quando existe duplicação em uma parte do código entre implementações de uma mesma abstração. Isso também pode ser motivado quando se perceber que em uma nova implementação será necessária parte da funcionalidade já presente em uma outra classe. Como exemplo, outro dia criei uma classe em que havia um código interno que validava se a ordem de invocação dos métodos estava correta, porém quando fui criar uma nova implementação percebi que a mesma validação seria necessária. Dessa forma, extraí um **Proxy** com a parte de validação da primeira classe e passei a utilizá-lo com todas as implementações.

A Figura 5.3 apresenta os passos da refatoração de extração de um **Proxy** de uma classe existente. O primeiro passo é criar a classe que representará o **Proxy**, delegando todas as chamadas dos métodos para a instância que está sendo encapsulada. Nesse ponto é importante fazer com que os testes não atuem isoladamente na classe, mas nela encapsulada pelo **Proxy** recém-criado. Inicialmente os testes deverão passar sem problemas, pois nenhum código foi adicionado na classe intermediária.

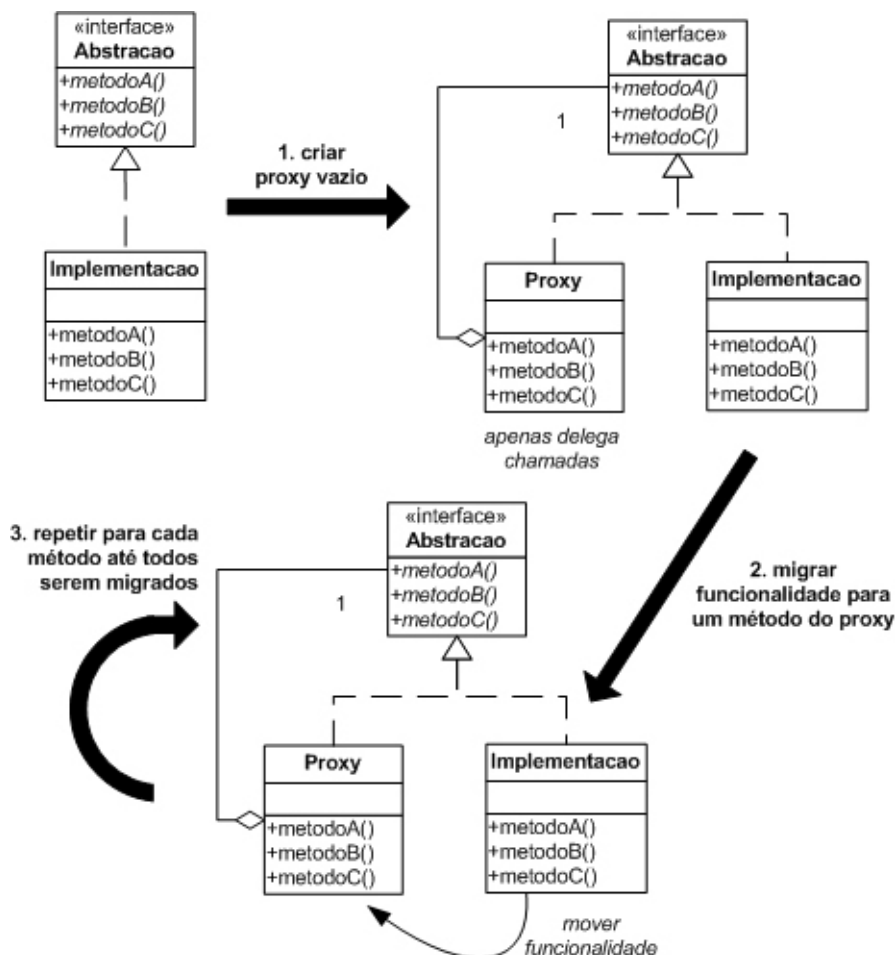



Figura 5.3: Refatoração de Extração de Proxy

Depois de possuir a suíte de testes atuando em cima da classe em conjunto com o `Proxy`, é hora de começar a migrar funcionalidade da classe encapsulada. A principal regra é que o comportamento conjunto deve permanecer inalterado. Dessa forma, a parte relativa a cada um dos métodos pode ir sendo movida para o intermediário, até que toda funcionalidade desejada esteja no `Proxy`. Depois disso, se o desenvolvedor achar necessário, ele pode separar os testes da classe dos testes relativos à funcionalidade do `Proxy`, que devem funcionar com qualquer outra implementação encapsulada.

Criando um Proxy no Eclipse

A criação da versão inicial do `Proxy`, que simplesmente delega os métodos para classe encapsulada, pode ser bem trabalhosa e braçal se a interface alvo possuir um número grande de métodos. Felizmente, o Eclipse possui uma funcionalidade de geração de código que cria automaticamente os métodos que delegam a chamada para os métodos de um atributo.

Para utilizá-la, primeiramente deve-se criar uma classe e incluir um atributo do tipo que será encapsulado. Não caia na tentação de adicionar a interface juntamente com seus métodos. Em seguida, clique com o botão direito sobre o código e escolha *Source > Generate Delegate Methods*. Nesse momento irá abrir uma janela apresentando quais métodos dos atributos você deseja criar e delegar para ele a chamada. É só escolher todos os métodos e em seguida declarar que a classe implementa a interface alvo. A Figura 5.4 mostra a caixa de diálogo aberta durante esse processo.

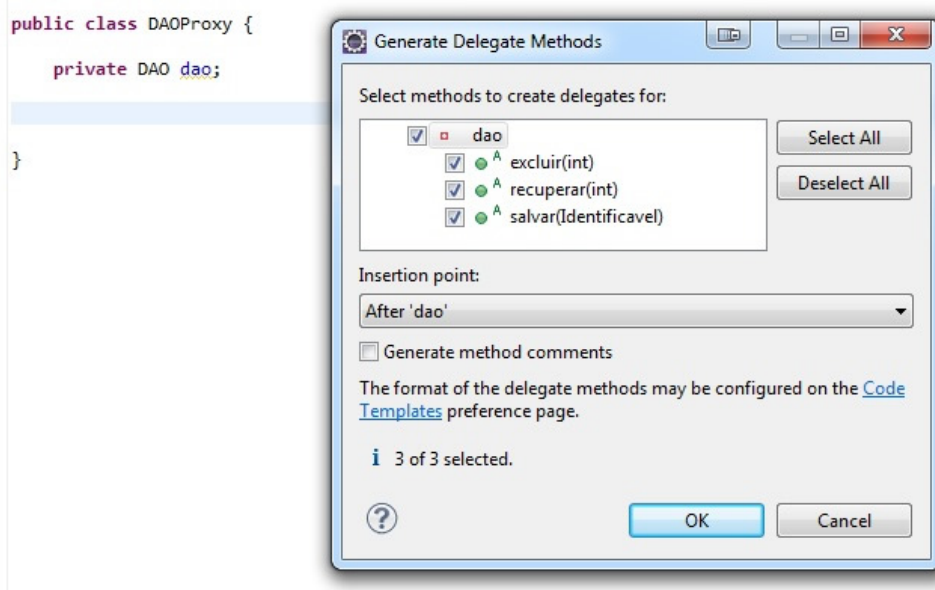



Figura 5.4: Automatizando a Delegação de Métodos no Eclipse

5.4 Adaptando Interfaces

"A incapacidade de se adaptar traz a destruição."
– Bruce Lee

Um outro cenário onde existe a necessidade de criar uma classe que envolva outra é quando possuímos uma classe que implementa uma interface, mas precisamos que ela implemente outra. Um exemplo do mundo físico ocorre quando compramos um notebook nos Estados Unidos e precisamos ligá-lo aqui nas tomadas do Brasil. No caso, o notebook americano possui uma fonte que possui a interface das tomadas americanas, porém precisamos que ele possua a que permita que ele seja ligado aqui no Brasil. Todos sabem que esse problema não é difícil de resolver, bastando um adaptador que encaixe em uma entrada no parão brasileiro e possua uma entrada no padrão americano. A Figura 5.5 mostra de forma visual a questão do adaptador.

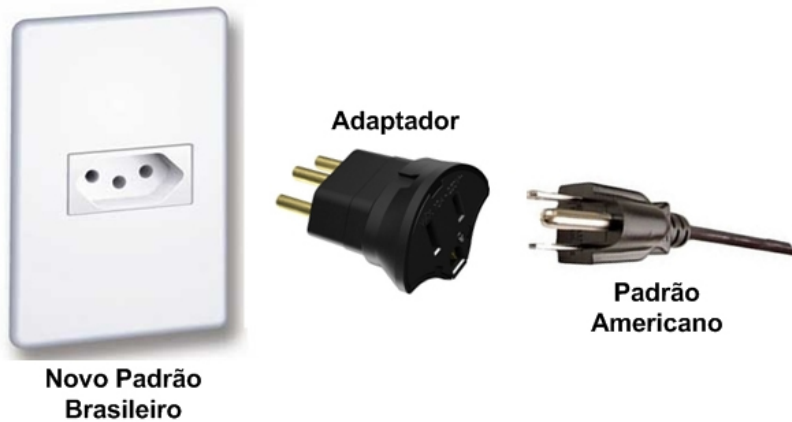



Figura 5.5: Representação do adaptador de tomadas

A ideia do padrão **`Adapter`** não é muito diferente de um adaptador de tomadas. Criamos uma classe que implementa a interface necessária, a qual precisamos nos adaptar, a "tomada nova". Internamente essa classe é composta por uma referência a um objeto que implementa a outra interface, a "tomada antiga". Dessa forma, o **`Adapter`** traduz as chamadas da interface que ele implementa para a classe que ele encapsula. Isso nem sempre é trivial, pois normalmente não é somente o nome do método que muda, mas outras questões muitas vezes inesperadas.

Estrutura do padrão Adapter

A estrutura do padrão **`Adapter`** é similar à dos padrões **`Proxy`** e **`Decorator`**, sendo que a principal diferença é que a classe que está sendo encapsulada não possui a mesma interface da que a está envolvendo. É justamente esse fato que permite a adaptação entre as interfaces. A Figura 5.6 mostra em um diagrama os principais participantes desse padrão.

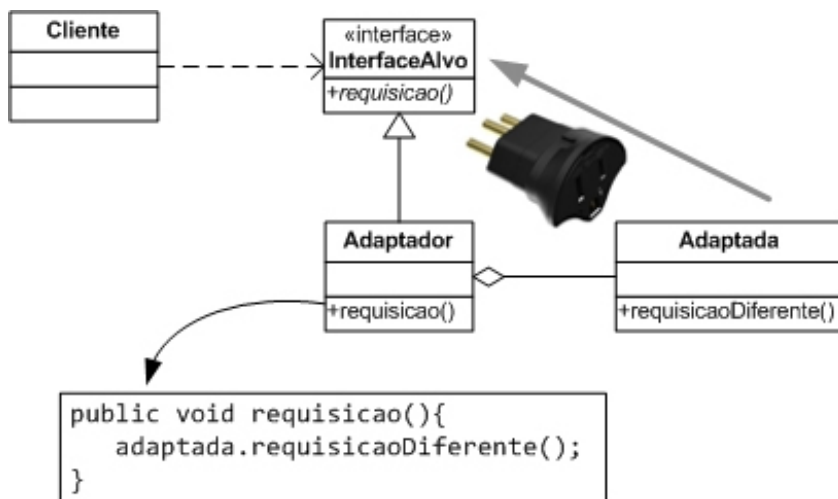



Figura 5.6: Estrutura do padrão Adapter

Nesse diagrama temos uma classe `Cliente` que depende de alguma forma de uma `InterfaceAlvo`. Sendo assim, para tornar possível ela acessar uma classe `Adaptada` que não implementa essa abstração, ela precisa utilizar a classe `Adaptador`, que irá receber chamadas de método conforme a `InterfaceAlvo` e irá chamar os métodos correspondentes na classe `Adaptada`.

É importante ficar claro que nem sempre a adaptação é simples de ser feita. Questões comuns que precisam ser traduzidas incluem a nomenclatura de classes e métodos, além da conversão de parâmetros e retornos. Questões mais complexas podem envolver diferentes formas de interagir com as classes. Por exemplo, o que é um parâmetro de método em uma pode ser um atributo passado no construtor na outra, ou ainda, o que é o retorno em uma pode ser um atributo populado em um parâmetro passado para o método na outra. Porém, a maior dificuldade é quando existem diferenças semânticas entre as diferentes APIs. Nesse caso, normalmente é necessária a ajuda de especialistas do domínio para compreender as diferenças e possibilitar a tradução.

Exemplo de Adaptador

Imagine uma aplicação que interaja com o serviço de SMS de operadoras de celular. Considere que a versão inicial da aplicação possua uma interface para interagir com o serviço de apenas uma operadora. A listagem a seguir apresenta a

interface SMSSender que é utilizada pela aplicação´para o envio de mensagens. Essa classe possui o método sendSMS() que recebe como parâmetro uma instância da classe SMS, também apresentada na listagem, e retorna um valor booleano dizendo se a mensagem foi enviada com sucesso ou não.

Listagem 5.7 - Interface Usada pelo Sistema para Envio de SMS:

```
public interface SMSSender {
    public boolean sendSMS(SMS sms);
}

public class SMS {
    private String destino;
    private String origem;
    private String texto;
    //getters e setters omitidos
}
```

Ao evoluir a aplicação, foi necessário incorporar o serviço de envio de SMS de uma outra operadora. Não foi uma surpresa muito grande quando foi constatado que a API para o acesso a funcionalidade era completamente diferente. A listagem com a interface dessa nova API, EnviadorSMS, está representada a seguir. A primeira diferença está nos parâmetros, que em vez de serem encapsulados em uma classe, no caso a SMS, são passado diretamente para o método. Outra diferença é que o texto da mensagem não é mais uma string, mas um array de strings. A primeira API recebia um texto longo e dividia em várias mensagens se necessário, porém nessa outra API a mensagem precisa ser divida em trechos de 160 caracteres antes da chamada do método. Finalmente, a nova API lança uma exceção para indicar uma falha, e não retorna um valor booleano para indicar o sucesso ou não.

Listagem 5.8 - Interface Usada pelo Sistema para Envio de SMS:

```
public interface EnviadorSMS {
    public void enviarSMS(String destino, String origem, String[] msgs)
        throws SMSException;
}
```

Para evitar que as classes cliente precisem ser alteradas e se acoplarem a uma nova API, decidiu-se criar um **Adapter** para traduzir as chamadas de uma interface para outra. Para criar esse adaptador é preciso lidar com todas as questões relativas

às diferenças entre as interfaces. A implementação do adaptador está representada na listagem a seguir.

Listagem 5.9 - Interface Usada pelo Sistema para Envio de SMS:

```
public class SMSAdapter implements SMSSender {

    private EnviadorSMS env;

    public SMSAdapter(EnviadorSMS env) {
        this.env = env;
    }
    public boolean sendSMS(SMS sms) {
        String dest = sms.getDestino();
        String orig = sms.getOrigem();
        String[] txts = quebrarMsgs(sms.getTexto());
        try {
            env.enviarSMS(dest, orig, txts);
        } catch (SMSException e) {
            return false;
        }
        return true;
    }
    private String[] quebrarMsgs(String text){
        int size = text.length();
        int qtd = (size%160==0)? size/160: size/160+1;
        String[] msgs = new String[qtd];
        for(int i=0; i<qtd; i++){
            int min = i*160;
            int max = (i==qtd-1)? size: (i+1)*160;
            msgs[i] = text.substring(min,max);
        }
        return msgs;
    }
}
```

O método `sendSMS()` inicialmente recupera as informações do objeto `SMS` e as atribui a variáveis locais. O método `quebrarMsgs()` é chamado para quebrar o texto da mensagem que está em apenas uma string em um array strings com tamanho máximo de 160 caracteres. Para lidar com a diferença do retorno, a chamada ao método `enviarSMS()` da classe `EnviadorSMS` é encapsulada em um bloco

`try/catch`. Dessa forma, caso a exceção for capturada o método retorna `false`, e caso o método siga seu fluxo normal, é retornado `true`.

Através desse exemplo, é possível observar como existem vários fatores que devem ser levados em consideração na hora da adaptação. Muitas vezes pode-se perder funcionalidade e informações. Em um caso oposto, podem ser fornecidas informações incompletas ou haver funcionalidades da interface que não podem ser executadas. Na classe `SMSAdapter`, por exemplo, para transformar a exceção recebida em um retorno booleano, perdeu-se a mensagem com a causa do erro e outras informações associadas a ela.

> **Adaptando Exceções**
>
> Quando a exceção lançada por um método precisa ser transformada na exceção lançada pela interface da classe adaptadora, a solução mais comum é realizar a chamada do método encapsulada em um bloco `try`, e lançar a nova exceção dentro do bloco `catch`. Caso ela seja criada sem relação com a anterior, o erro lançado possuirá em seu *stacktrace* o método do **`Adapter`** como origem, omitindo a real causa do erro. O segredo nesse caso é não esquecer de passar a exceção original no construtor da nova, dessa forma o *stacktrace* anterior será incluído como causa do erro.

Adaptadores para Migração de APIs

Um outro cenário onde o padrão **`Adapter`** costuma ser bastante utilizado é para dar suporte a classes que utilizaram uma versão antiga de uma API em uma versão nova do software. Imagine, por exemplo, que em uma aplicação fossem criadas classes que implementassem a interface de um framework para serem utilizadas por ele. Caso o framework modifique essa interface em sua próxima versão, a aplicação não poderá utilizar as classes já criadas. Isso é um grande problema, visto que a aplicação pode nem compilar com a nova versão do framework.

Como uma forma de contornar esse problema, o framework pode prover um adaptador que adapte as classes desenvolvidas na versão anterior da interface para a versão nova. Dessa forma, as classes da aplicação podem usar esse adaptador para continuar utilizando as classes já desenvolvidas.

Como um exemplo disso, a JDK 1.0 possuía a interface `Enumeration`, com os métodos `hasMoreElements()` e `nextElement()`, e era implementada por classes que desejavam permitir a iteração de seus elementos. Na JDK 1.2 foi introduzida a interface `Iterator`, com o mesmo objetivo, porém com nomes de métodos menores, `hasNext()` e `next()`, e com um método opcional `remove()`.

Para permitir que códigos que trabalhem com uma interface possam lidar com classes que sabem trabalhar com a outra, a biblioteca de classes *Apache Commons Collections* provê a classe `IteratorUtils`. Ela possui os métodos estáticos `asIterator()`, que adapta um `Enumeration` para um `Iterator`, e `asEnumeration()`, que faz a adaptação inversa. Dessa forma, se um código que utiliza `Enumeration` precisa utilizar uma classe que implementa `Iterator`, ele pode utilizar o `IteratorUtils` para fazer essa adaptação.

5.5 Considerações Finais do Capítulo

Este capítulo abordou padrões que definem classes que envolvem objetos, servindo como intermediárias entre a que fornece a funcionalidade e a cliente. Com o uso do encapsulamento e do polimorfismo, é possível tornar essa intermediária transparente, permitindo facilmente a adição de uma nova camada entre elas. Esse recurso poderoso permite a utilização de diversos artifícios escondidos sob a máscara da interface da qual a classe cliente depende.

Na primeira parte do capítulo, foram abordados os padrões **`Proxy`** e **`Decorator`**, nos quais essa classe intermediária possui a mesma abstração da que a está envolvendo. Dessa forma, ela pode adicionar novas funcionalidades e se passar pela própria classe. Já na segunda parte, foi abordado o padrão **`Adapter`** no qual a intermediária é utilizada para encapsular uma classe com uma interface diferente da sua. Nesse caso, a ideia é justamente permitir que uma classe com uma determinada interface possa ser utilizada por outra que sabe interagir com classes com interface diferente.

Capítulo 6

Estratégias de Criação de Objetos

"A verdadeira felicidade vem da alegria de atos bem feitos, o entusiasmo de criar coisas novas."
– Antoine de Saint-Exupery

Nos capítulos anteriores, foram apresentados diversos padrões que montam estruturas utilizadas para fornecer flexibilidade e extensibilidade nas classes da aplicação. Quando utilizamos herança, devemos escolher entre diferentes implementações de uma abstração para que ela tenha o comportamento adequado. Com composição, outros objetos devem ser instanciados e inseridos na classe principal. Com a composição recursiva, diversas instâncias de implementações de uma abstração são combinadas em uma estrutura que fornece o comportamento desejado. Finalmente, para a utilização de `Proxies`, é preciso envolver o objeto e colocá-lo no lugar do encapsulado.

Para os padrões funcionarem é preciso que essas estruturas estejam montadas, porém saber como criá-las de forma adequada não é tão simples. Por exemplo, de nada adianta a classe utilizar as abstrações para se desacoplar das implementações,

se na hora de criar os objetos ela referencia diretamente as classes. Os objetos devem estar desacoplados também no momento da criação! Não somente a utilização de uma estrutura de herança ou composição deve ser transparente à classe cliente, mas também a criação dessa estrutura. Para que realmente exista flexibilidade, deve ser possível criar as diferentes implementações ou combinações de classe para que o comportamento possa ser alterado.

Esse capítulo irá abordar os padrões para a criação de objetos. Eles mostram técnicas para o desacoplamento da lógica de criação de objetos, principalmente quando precisa-se utilizar um dos padrões apresentados nos capítulos anteriores. Também serão apresentadas soluções para encapsular uma lógica complexa de criação e para vincular a criação de objetos relacionados. A próxima seção inicia o capítulo mostrando por que os construtores, a forma tradicional de criar objetos, podem não ser suficientes para encapsular essa lógica de criação.

6.1 Limitações dos Construtores

Se você perguntar para qualquer desenvolvedor Java como se cria um objeto, ele irá prontamente responder que é através de construtores. Apesar de realmente não ser possível criar uma nova instância de uma classe sem a invocação de um construtor, sua aplicação não precisa interagir diretamente com eles para criar os objetos de que precisa. Principalmente quando se deseja desacoplar a classe cliente da que está sendo criada. Se o construtor for diretamente invocado, o código da classe cliente precisará ser mudado para substituí-la por uma nova implementação.

Esta seção mostra algumas limitações dos construtores que dificultam sua utilização para a criação de objetos. Algumas dessas questões são abordadas no famoso livro *Effective Java* de Joshua Bloch [3].

Dois Construtores Não Podem Ter Parâmetros de Mesmo Tipo

Uma característica dos construtores é que eles possuem o mesmo nome da classe que criam. Isso não é um problema se ela possui um processo de criação simples, fornecendo um ou dois construtores para sua criação. Porém, quando existem diversas formas de criar objetos de uma mesma classe, isso pode ser um problema. Para começar, toda a expressividade que é possível através da utilização de nomes descritivos para métodos não pode ser utilizada nos construtores, visto que eles precisam ter o mesmo nome da classe. Isso dificulta bastante quando uma classe possui diversos construtores e não se sabe exatamente qual o comportamento que será fornecido

por cada um deles.

Essa questão fica ainda mais crítica quando é necessária a criação de construtores que possuem parâmetros de mesmo tipo. Como isso não é possível na linguagem Java, é preciso utilizar artifícios para diferenciar os construtores. Por exemplo, adicionar um parâmetro diferente a mais para diferenciá-los, ou ainda criar apenas um construtor com um parâmetro dizendo qual o tipo de criação que se deseja.

Para ficar mais concreta essa situação, imagine que uma classe chamada `CoordenadaGeografica` possa receber strings com representações textuais das coordenadas em seu construtor. Como existem diversos formatos para coordenada, como o **Geodésico** (*Lat 021°30.4423' S ; Long 055°09.6734' W*) e o **Geodésico Decimal** (*Lat -21.5070334899 ; Long -55.4119080998*), a intenção seria ter construtores separados para cada tipo de formato. Infelizmente, como ambos construtores teriam o mesmo nome e receberiam uma string como parâmetro, essa solução não seria possível de ser implementada.

Um Construtor Sempre Cria um Novo Objeto

Uma outra característica de um construtor é que toda vez que ele é chamado ele retorna um novo objeto da classe. Certamente é isso mesmo que um construtor deveria fazer, porém muitas vezes deseja-se utilizar um objeto que já existe e não gerar um novo. Se esse for o caso, fica inviabilizada a criação desses objetos através de construtores.

Objetos que não contenham estado, ou seja, contenham basicamente métodos com lógica de negócio, são bons candidatos para terem um número limitado de instâncias. Outras vezes, deseja-se manter uma única entidade para representar uma determinada entidade em todo sistema. Imagine que em um sistema de locadora de carros deseja-se ter uma única instância para representar a instância de `TipoCarro` que representa um *Corsa*. Dessa forma, quando for necessário representar um *Corsa* em vez de se criar uma nova instância, a já existente deve ser retornada. Nada disso pode ser feito utilizando construtores diretamente.

Um Construtor Só Pode Retornar Objetos da Mesma Classe

Talvez o principal problema dos construtores seja retornar objetos apenas da classe que definiu aquele construtor. Não é possível, por exemplo, retornar uma instância de uma subclasse ou envolver o objeto em um **`Proxy`**. Dessa forma, a classe que invoca o construtor de uma outra se acopla a ela fortemente. Por mais que

ela utilize a abstração em outras partes do código, será necessário uma alteração no código de criação caso uma outra classe precise ser utilizada.

Para entender como isso pode ser um problema, imagine uma classe que é utilizada em vários pontos do sistema. Suponha agora que ela possuísse um grande método que fosse refatorado para a utilização do padrão **`Template Method`**, no qual subclasses implementariam partes do algoritmo. Uma situação similar seria se fosse criado um **`Decorator`** para adicionar alguma nova funcionalidade a essa classe. Se ela fosse utilizada em vários pontos diferentes do sistema, o código de criação deveria ser modificado em todos esses pontos para que a subclasse adequada fosse criada ou que o **`Decorator`** fosse criado para encapsular a instância.

Sendo assim, é possível perceber que a utilização direta de construtores pode dificultar bastante o uso dos padrões que foram vistos até o momento nesse livro. A lógica de criação pode se complicar principalmente quando alguns padrões são combinados e surgem questões como: escolher a subclasse correta para instanciar; escolher as implementações para compor a instância; e adicionar funcionalidades envolvendo o objeto em **`Proxies`**. As próximas seções apresentam alguns padrões que podem ser utilizados para eliminar essas limitações e encapsular uma lógica complexa de criação de objetos.

6.2 Criando Objetos com Métodos Estáticos

"Se permaneces estático na derrota, mova-se rumo à vitória."
– Edilson Sanches Pontes

A solução mais simples para eliminar a necessidade da utilização de construtores diretamente é a criação de métodos estáticos para criarem os objetos. Essa prática é conhecida como **`Static Factory Method`**. Apesar de ser considerado um padrão, essa prática não foi devidamente documentada como um padrão e é abordada com mais detalhes no *Effective Java* [3].

Nesse padrão, as classes clientes delegam para os métodos estáticos a lógica de criação das instâncias. Esses métodos recebem os parâmetros necessários e estes retornam a instância adequada. Como esses métodos de criação não possuem as restrições dos construtores, eles podem resolver as limitações descritas na seção anterior.

Um dos primeiros benefícios da utilização do **`Static Factory Method`** está na expressividade, pois é possível nomear o método de acordo com a lógica de criação envolvida. Isso acaba resolvendo o problema dos construtores

com tipos de parâmetro repetidos, pois é só criar nome diferentes para eles. No exemplo da classe `CoordenadaGeografica` citado anteriormente, seria possível possuir métodos de criação chamados `criarCoordenadaGeodesico()` e `criarCoordenadaGeodesicoDecimal()`. Dessa forma, o código cliente ficaria inclusive mais claro em relação à estratégia de criação que está sendo utilizada.

Outra possibilidade interessante é a de retornar um objeto já existente. Esse objeto normalmente é guardado na primeira vez que ele é criado e depois é aproveitado para as chamadas seguintes. Ele é armazenado em uma variável estática, ou em alguma coleção também em uma variável estática. Um exemplo clássico disso é a obtenção de conexões para um banco de dados. A classe `DriverManager`, por exemplo, disponibiliza o método estático `getConnection()` em que uma conexão é retornada. Algumas implementações criam um *pool* de conexões, onde as conexões são reaproveitada depois de serem utilizadas.

Um outro ponto interessante do **`Static Factory Method`** é que ele pode retornar qualquer implementação. Dessa forma, é possível introduzir novas subclasses e retorná-las nesse método de forma transparente à classe cliente. Assim, a classe cliente dependente realmente apenas do tipo retornado pela pelo método fábrica e não das implementações retornadas. Isso simplifica muito refatorações que criam novas classes de uma abstração, como a criação de **`Template Method`** e de um **`Proxy`**, pois ao alterar o **`Static Factory Method`**, todas as que o utilizam para a criação poderão receber uma nova implementação de forma transparente.

Static Factory Method X Factory Method

Antes de ir fundo nas características do **`Static Factory Method`** é importante deixar bem claro que ele é diferente do padrão **`Factory Method`** apresentado no Capítulo 2. O padrão que está sendo descrito nessa seção consiste em um método estático utilizado para encapsular a criação e/ou obtenção de instâncias por parte da classe cliente. Já o **`Factory Method`** é um *hook method* definido por uma classe para delegar a criação de uma instância para suas subclasses.

Exemplos de Static Factory Method

Para exemplificar a utilização de métodos fábrica, vamos retomar o exemplo do gerador de arquivo apresentado nos capítulos anteriores (a versão que utiliza o pa-

drão **Composite**). A listagem a seguir apresenta o código de uma classe que disponibiliza **Static Factory Methods** para a criação de geradores de arquivo. Nessa implementação optou-se por disponibilizar métodos diferentes para a criação dos geradores de arquivos em XML e de propriedades. Outra alternativa seria a passagem de um parâmetro que identificaria o formato do arquivo.

Listagem 6.1 - Fábrica de GeradorArquivo utilizando Static Factory Method:

```
public abstract class FabricaGerador {

    public static final String ZIP = "ZIP";
    public static final String CRYPTO = "CRYPTO";

    public static GeradorArquivo criarGeradorXML
                                (String... processadores) {
        GeradorArquivo g = new GeradorXML();
        g.setProcessador(criarProcessador(processadores));
        return g;
    }
    public static GeradorArquivo criarGeradorPropriedades
                                (String... processadores) {
        GeradorArquivo g = new GeradorPropriedades();
        g.setProcessador(criarProcessador(processadores));
        return g;
    }
    private static PosProcessador criarProcessador
                                (String... processadores) {
        if(processadores.length > 1) {
            PosProcessadorComposto pp = new PosProcessadorComposto();
            for(String proc : processadores){
                pp.add(criarProcessador(proc));
            }
            return pp;
        } else if(processadores[0].equals(ZIP)){
            return new Compactador();
        } else if(processadores[0].equals(CRYPTO)){
            return new Criptografador();
        }
    }
}
```

Para os processadores, é passado um array de strings que identificam os pós-

processadores em sua respectiva ordem. Ambos os métodos fábrica delegam a criação deles para o método `criarProcessador()`. Observe que caso exista mais de um, a classe `PosProcessadorComposto` que implementa o padrão **Composite** é utilizada.

O código a seguir mostra como a classe `FabricaGerador` seria utilizada na criação de um `GeradorArquivo`. Um dos métodos estáticos, como `criarGeradorXML()` ou `criarGeradorPropriedades()`, deve ser invocado passando como parâmetro de forma opcional as strings que representam os pós-processadores. Note que a classe tem contato apenas com a `FabricaGerador` e a abstração `GeradorArquivo`, sendo que o uso de todas as subclasses e pós-processadores ficam encapsulados dentro da fábrica.

Listagem 6.2 - Exemplo de Criação do GeradorArquivo:

```
GeradorArquivo ga = FabricaGerador.criarGeradorXML(
    FabricaGerador.ZIP,FabricaGerador.CRYPTO);
```

A própria API do Java utiliza esse padrão em diversos pontos para encapsular a criação de objetos. Um exemplo comum são os métodos `parseInt()` e `valueOf()` da classe `Integer`. Além de serem métodos estáticos que retornam instâncias de `Integer`, eles fazem cache para que a mesma instância seja retornada caso o mesmo número seja passado. Isso ocorre igualmente com métodos similares em outras classes que encapsulam tipos primitivos.

Impedindo de Invocar o Construtor

Disponibilizar um **Static Factory Method** não impede as classes clientes de invocarem o construtor e ignorar toda a lógica de criação encapsulada. Para resolver essa questão, o segredo é fazer com que o construtor tenha uma visibilidade que dê acesso ao método fábrica e não dê acesso às classes clientes. Se o método estiver na mesma classe que está sendo criada, o construtor pode ser declarado como privado. Se o método fábrica estiver em uma classe diferente, então uma solução é declará-lo como protegido e colocar essa classe fábrica no mesmo pacote que as que são criadas por ele.

6.3 Um Único Objeto da Classe com Singleton

"Só pode haver um!"

– Connor MacLeod, Highlander

Um caso especial da utilização do **Static Factory Method** é quando se deseja que a aplicação possua apenas uma instância de uma determinada classe. Isso normalmente é motivado por questões de negócio, em que faz sentido apenas um objeto daquele tipo. Por exemplo, imagine um software que represente um jogo de xadrez entre duas pessoas. Nesse contexto provavelmente fará sentido apenas um tabuleiro de jogo. A estrutura para permitir esse tipo de construção é o padrão **Singleton**.

A listagem a seguir apresenta um exemplo de implementação do padrão **Singleton**. No exemplo, o **Singleton** é utilizado em uma classe que representa e armazena configurações do sistema. Esse é um exemplo em que o uso desse padrão é adequado, pois só existe uma única configuração para todo o sistema, e dessa forma ela pode ser facilmente obtida a partir de qualquer classe.

Listagem 6.3 - Exemplo de Singleton:

```
public class Configuracao {

    private static Configuracao instancia;

    public static Configuracao getInstancia() {
        if(instancia == null)
            instancia = new Configuracao();
        return instancia;
    }

    // Construtor privado!
    private Configuracao() {
        //lê as configurações
    }

    // ...
}
```

Nesse caso, o método fábrica é normalmente definido na própria classe. O artifício de definir um construtor privado é frequentemente utilizado para impedir a criação de outras instâncias da classe. Um atributo estático é definido para armazenar essa instância e um método estático é disponibilizado de forma pública para

sua recuperação. Observe que na listagem de exemplo o objeto é criado no primeiro acesso ao método. Uma implementação alternativa, seria incluir essa criação em um bloco estático para que ela fosse realizada quando a classe fosse carregada pela máquina virtual.

A listagem a seguir mostra o exemplo de como uma classe pode obter a instância da que implementa o padrão **Singleton**. Observe que o processo não é muito diferente da invocação de um **Static Factory method**. Em qualquer local da aplicação em que o método `getInstancia()` for invocado, o mesmo objeto será retornado.

Listagem 6.4 - Obtenção de um Singleton:

```
Configuracao c = Configuracao.getInstancia();
```

Uma das vantagens do **Singleton** é a facilidade de acesso dessa instância por qualquer objeto na aplicação. De qualquer classe é possível chamar o método `getInstancia()` e obtê-la. Isso evita, por exemplo, que essa instância única precise ser passada como parâmetro para diversos lugares. No exemplo do tabuleiro de xadrez, essa provavelmente seria uma classe que precisaria estar acessível de diversos locais de acordo com a lógica do jogo.

Uma solução utilizando um **Singleton** oferece uma flexibilidade muito maior do que uma solução que utiliza métodos estáticos para a execução das regras de negócio. Por ser um objeto, a instância única pode ser especializada e encapsulada por um **Proxy**, o que não é possível fazer com métodos estáticos. Essa questão prejudica bastante a testabilidade de classes que acessam métodos estáticos, pois não é possível substituí-los por um objeto falso com propósitos de teste, conhecido como Mock Object [25]. Uma solução que concilia a praticidade dos métodos estáticos com a flexibilidade do **Singleton** seria os métodos estáticos delegarem a lógica de sua execução para os métodos do **Singleton**.

O Lado Negro do Singleton

O padrão **Singleton** deve ser utilizado com muito cuidado e nas situações em que realmente fizer sentido ter apenas uma instância de uma classe. Muitos acham que, por ser um padrão, ele pode ser utilizado em qualquer parte do sistema. O resultado é que o **Singleton** acaba sendo usado como uma variável global da orientação a objetos, o que pode reduzir a flexibilidade da aplicação deixando sua modelagem deficiente. Por ele ser utilizado mais em situações inadequadas, muitos acabam considerando o **Singleton** como uma má prática. Sendo assim, toda vez

que for utilizar um **Singleton** reflita bastante se sua utilização é mesmo necessária. Também veremos, em um capítulo posterior, que inversão de controle e injeção de dependências podem ajudar bastante nesse caso.

6.4 Encapsulando Lógica Complexa de Criação com Builder

"Podemos construir? Sim! Podemos sim!"
– Bob, o Construtor

Pelo que já foi dito sobre a criação de objetos até o momento nesse capítulo, acredito que tenha sido possível perceber como a criação de determinados tipos pode ser complexa. Nesses casos, a lógica da criação, que muitas vezes precisa validar parâmetros ou buscar informações em arquivos, acaba se misturando com a lógica da própria classe. Sendo assim, mantê-las na mesma classe pode deixá-la grande e confusa, tanto com a utilização de construtores quanto com a utilização de métodos estáticos de criação.

O padrão **Builder** fornece uma solução para essa questão da criação de objetos complexos, com a definição de uma classe responsável pelo processo de criação. A partir de um **Builder** é possível seguir o mesmo processo de criação para diferentes estruturas e diferentes representações. Dessa forma, a classe que invoca os métodos de um **Builder** para a criação do objeto que precisa fica desacoplada dessa lógica de criação complexa e da classe concreta que está sendo criada.

Além disso, pode ser criada uma abstração para um **Builder** de forma que diferentes implementações possam ser criadas. Desse modo, os clientes podem guiar a criação de diferentes objetos através do **Builder**, porém sem saber a implementação que está sendo utilizada e, consequentemente, qual a classe do objeto que está sendo criado. Esse tipo de prática é muito comum em APIs que disponibilizam interfaces que são implementadas por diferentes fornecedores. Por exemplo, imagine uma API que lida com certificados digitais. O padrão **Builder** pode ser utilizado para criar o objeto que assina um arquivo digitalmente, porém cada fornecedor possuirá uma implementação diferente do **Builder** de acordo com suas classes.

A Figura 6.2 apresenta a estrutura do padrão **Builder**. No diagrama, a classe `Cliente` precisa criar uma instância da abstração `Produto` e utiliza um `Builder` para essa tarefa. Observe que pela estrutura é possível ter várias implementações do `Builder`, possibilitando que diferentes estratégias de implementação sejam utiliza-

das. Porém é importante ressaltar que essa abstração do **`Builder`** só deve ser usada quando realmente for necessária.

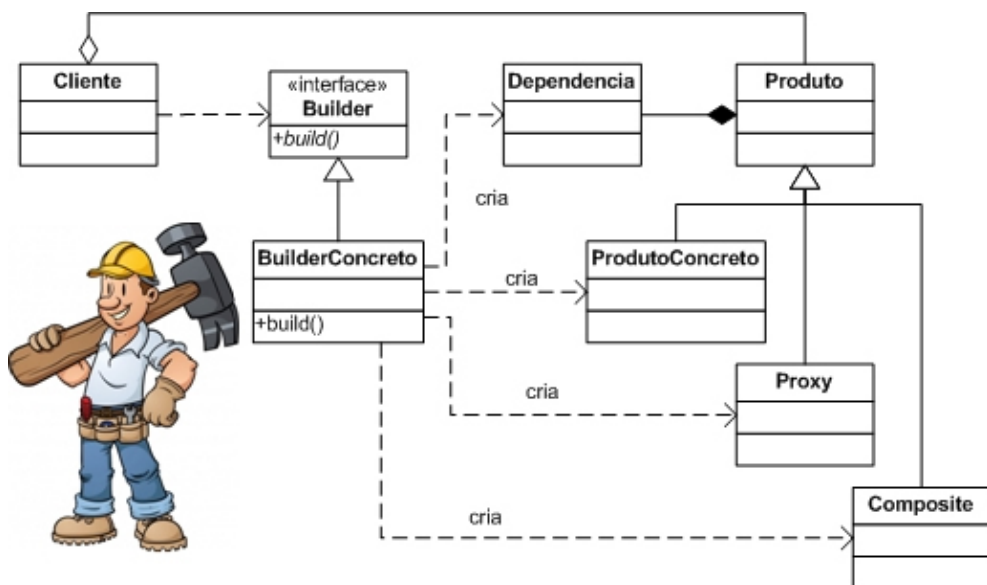



Figura 6.1: Representação do padrão Builder

Na figura alguns outros padrões estão representados para ilustrar a complexidade que o processo de criação pode possuir. Veja que a classe representada como `Produto` pode possuir subclasses, pode ser composta por outras, pode ser envolvida por um **`Proxy`** e pode ter suas implementações combinadas por um **`Composite`**, isso entre outros padrões que podem ser utilizados. Nenhum desses outros padrões citados são obrigatórios para a utilização de um **`Builder`**, porém sua presença aumenta a complexidade do processo de criação favorecendo a utilização de um padrão de criação de objetos.

Criando o SessionFactory do Hibernate com um Builder

Para melhorar a compreensão do padrão **`Builder`** essa seção apresenta um exemplo do seu uso no framework Hibernate para a criação do objeto da classe `SessionFactory`. O Hibernate é um framework cuja principal funcionalidade é realizar o mapeamento objeto-relacional, ou seja, entre classes e tabelas do banco de dados. Para que isso possa ser feito, é preciso que diversas configurações sejam lidas

de arquivos e de anotações em classes, tornando a criação de um `SessionFactory`, a classe que gera as sessões para o acesso a base de dados, um processo bem complexo.

A listagem a seguir mostra a utilização da classe `AnnotationConfiguration` para a criação de um `SessionFactory`. Nesse contexto, `AnnotationConfiguration` é a implementação do padrão **`Builder`**. Observe como diversos método são chamados para adicionar configurações em relação a como essa classe deve ser criada. O encadeamento na chamada dos método se deve ao fato de cada um deles retornar a própria instância de `AnnotationConfiguration`, com exceção do método `buildSessionFactory()` que retorna a instância de `SessionFactory`.

Listagem 6.5 - Criação de um SessionFactory do Hibernate:

```
SessionFactory sessionFactory = new AnnotationConfiguration()
    .addPackage("com.lojavirtual")
    .addAnnotatedClass(CarrinhoCompras.class)
    .addAnnotatedClass(Cliente.class)
    .addAnnotatedClass(Produtos.class)
    .addResource("orm.xml")
    .configure()
    .buildSessionFactory();
```

Esse é um bom exemplo de uso do padrão **`Builder`** pois envolve realmente um processo complexo de criação. A partir da leitura de arquivos XML e das anotações das classes mapeadas são extraídas as informações necessárias para o `SessionFactory`. É importante ressaltar que essa lógica extrapola as responsabilidades da classe `SessionFactory`, fazendo todo sentido a separação da responsabilidade da sua criação em uma classe separada.

Builder com Interface Fluente

O termo **interface fluente** foi cunhado por Martin Fowler e Erick Evans [14], como uma forma de descrever um estilo de construção de interfaces. A ideia é dar o nome dos métodos da classe de forma que o código pareça uma frase em linguagem natural. A listagem a seguir mostra o exemplo de uma interface tradicional, utilizando métodos começados com `get` e `set`, e a mesma classe utilizando uma interface fluente. Com uma interface fluente é como se você estivesse definindo uma nova linguagem dentro da linguagem de programação a partir dos nomes

dos métodos, prática que também é chamada de DSL (Linguagem Específica de Domínio) interna [15].

Listagem 6.6 - Exemplo de uso de interface fluente:

```
//Estilo comum
Pessoa p = new Pessoa();
p.setNome("João");
p.setDataNascimento("30/03/1985");
p.setCargo("gerente");

//Interface fluente
Pessoa p = new Pessoa()
    .chamada("João")
    .nascidaEm("30/03/1985")
    .comCargo("gerente");
```

A parte mais importante de uma interface fluente está na nomenclatura utilizada para os métodos. Eles devem soar como se fossem parte de uma frase, de forma que "*uma nova pessoa chamada João*" vira `new Pessoa().chamada("João")`. Outra questão é que os métodos devem retornar a instância da própria classe, ou seja, `this`, para as chamadas de método possam ser encadeadas. No exemplo apresentado, as chamadas dos métodos `chamada()`, `nascidaEm()` e `comCargo()` retornam a própria instância de `Pessoa`.

Muitas vezes, as aplicações precisam de utilizar as interfaces no padrão Java Beans (utilizando métodos com `get` e `set`), por requisito dos frameworks utilizados. Isso acaba inviabilizando a utilização da interface fluente nas classes de domínio. Porém, isso não impede que elas sejam utilizadas nos **`Builders`**, que criam essas classes. É importante mais uma vez ressaltar que, para que a utilização do **`Builder`** seja adequada, é preciso que o processo de criação do objeto em questão seja complexo e exija a configuração de diversos parâmetros.

Um dos frameworks pioneiros na utilização de interfaces fluentes para a criação de objetos foi o framework JMock [24]. Ele é utilizado para a criação de mock objects, que são objetos falsos utilizados para substituir as dependências de uma classe em testes de unidade. A listagem a seguir apresenta alguns trechos de código do JMock para definir o comportamento do objeto falso e suas expectativas em relação às chamadas da classe testada. Observe que apesar das limitações relativas à sintaxe da linguagem Java, é possível ler o código como se fosse uma frase.

Listagem 6.7 - Interface Fluente no JMock:

```
//uma chamada ao método log() com uma string contendo 'erro'
one (logger).log(with(stringContaining("error")));

//exatamente 3 chamadas de count()
//retornando consecutivamente 10, 20 e 30
exactly(3).of(counter).count();
will(onConsecutiveCalls(returnValue(10),
  returnValue(20),returnValue(30)));

//permitir chamada ao método sqrt com valor menor que zero
//irá lançar a exceção IllegalArgumentException
allowing (calculator).sqrt(with(lessThan(0)));
will(throwException(new IllegalArgumentException());
```

Exemplos de Utilização do Builder

Para exemplificar os conceitos do padrão **Builder** e de interface fluente, será utilizado como contexto o mesmo exemplo para a criação da classe `GeradorArquivo` mostrado com o **Static Factory Method**. A listagem a seguir apresenta a implementação da classe `BuilderGerador`. Essa classe possui um atributo `instancia` que armazena o objeto que está sendo construído. Ao chamar um dos dois métodos `gerandoEmXML()` ou `gerandoEmPropriedades()` a subclasse correta é recuperada e atribuída ao atributo. Já os métodos `comCriptografia()` e `comCompactacao()` adicionam os processadores na instância sendo construída, utilizando o **Composite** quando houver mais de um. No final de todo processo, o método `construir()` deve ser chamado para retornar o objeto criado.

Listagem 6.8 - Builder para a classe GeradorArquivo:

```
public class BuilderGerador {

    private GeradorArquivo instancia;

    public BuilderGerador gerandoEmXML() {
        instancia = new GeradorXML();
        return this;
    }
```

```
    public BuilderGerador gerandoEmPropriedades() {
        instancia = new GeradorPropriedades();
        return this;
    }
    public BuilderGerador comCriptografia() {
        adicionaProcessador(new Criptografador());
        return this;
    }
    public BuilderGerador comCompactacao() {
        adicionaProcessador(new Compactador());
        return this;
    }
    private void adicionaProcessador(PosProcessador p) {
        PosProcessador atual = instancia.getProcessador();
        if(atual instanceof NullProcessador) {
            instanciasetProcessador(p);
        }else{
            PosProcessadorComposto pc = new PosProcessadorComposto();
            pc.add(atual);
            pc.add(p);
            instancia.setProcessador(pc);
        }
    }
    public GeradorArquivo construir() {
        return instancia;
    }
}
```

Para deixar o exemplo mais interessante, vamos imaginar que existam ainda **Proxys** que possam ser utilizados para encapsular a classe `GeradorArquivo`. Dentro desse exemplo, imagine que a classe `ProxyAssincrono` invoque o método de geração de arquivos em uma thread diferente e que a classe `LoggerProxy` grave as chamadas realizadas em um arquivo de log para fins de auditoria. A listagem a seguir mostra como essas classes poderiam facilmente ser incorporadas no **Builder**. Ao ser chamado o método, o proxy encapsula a instância armazenada e o resultado é atribuído a ela mesma.

Listagem 6.9 - Adicionando métodos no Builder para adição de Proxies:

```
public class BuilderGerador {
    //...
    public BuilderGerador assincrono() {
```

```
        instancia = new ProxyAssincrono(instancia);
        return this;
    }
    public BuilderGerador registrandoAuditoria() {
        instancia = new LoggerProxy(instancia);
        return this;
    }
}
```

Para ver como fica a construção do objeto utilizando interface fluente, a listagem a seguir mostra um exemplo de código. Observe como a construção do objeto flui de forma fácil entre as chamadas de método para sua configuração.

Listagem 6.10 - Exemplo de utilização do Builder com interface fluente:

```
GeradorArquivo ga = new BuilderGerador().gerandoEmXML()
    .comCriptografia().comCompactacao().assincrono()
    .registrandoAuditoria().construir();
```

6.5 Relacionando Famílias de Objetos com Abstract Factory

"Família que ${algumVerbo} unida, permanece unida."
– Dito popular

Nos padrões anteriores, foram abordados problemas referentes à complexidade de criação dos objetos, além de seu desacoplamento da classe cliente e da própria classe que está sendo criada. Um outro problema surge quando mais de um objeto relacionado precisa ser criado. Muitas vezes existem famílias de objetos nos quais existem classes relacionadas em hierarquias paralelas. Um exemplo é quando uma mesma API é implementada por diversos fornecedores. Nessa caso, cada um deles fornece implementações diferentes para as mesmas abstrações. Não faz sentido utilizar uma classe de um fornecedor junto com outra classe de outro, pois apesar de obedecerem às mesmas abstrações elas não podem ser misturadas.

Um padrão que existe que foca na solução desse problema é o **`Abstract Factory`**. Nesse padrão, em vez de termos uma fábrica para a criação de um objeto, ela é destinada à criação de uma família de objetos relacionados. Dessa forma, se todos os objetos relacionados forem obtidos a partir da mesma fábrica, não haverá inconsistência entre eles.

Um exemplo que temos na plataforma Java é a API JDBC. A classe `Connection` representa uma conexão com o banco de dados e também é utilizada para criar outros objetos utilizados para interagir com o banco de dados. Se misturarmos os objetos obtidos de conexões diferentes (o `ResultSet` do MySQL com o `Statement` do PostGreSQL, por exemplo) obteremos um erro, porém o fato de existir uma instância como um ponto central de onde todos podem ser obtidos evita que esse tipo de erro aconteça. Nesse caso, a classe `Connection` faz o papel de **Abstract Factory**, pois só vai fabricar objetos relativos ao banco de dados com o qual ela sabe se comunicar.

Estrutura do Abstract Factory

A estrutura básica do padrão **Abstract Factory** está apresentada no diagrama a seguir. As classes `ProdutoA` e `ProdutoB` representam as classes pertencentes a uma família. Dessa forma, a família "X" é representada pelas implementações `ProdutoAX` e `ProdutoBX` e o similar ocorre com a família "Y". O exemplo apresenta apenas dois tipos de produto e duas famílias, porém em uma implementação desse padrão pode haver mais de cada um deles.

A classe `FabricaAbstrata` é a que abstrai a criação de uma família de objetos. Cada implementação é responsável pela criação de uma das famílias de objeto e possui métodos para a criação de cada um dos tipos de objeto da família. Como pode ser visto no diagrama, cada família possui uma implementação da **Abstract Factory**. Por exemplo, a `FabricaFamiliaX` cria instâncias das classes `ProdutoAX` e `ProdutoBX`.

Existe uma troca importante que se faz quando se utiliza o **Abstract Factory**. Utilizando esse padrão, é muito fácil criar e introduzir na aplicação uma nova família de objetos. Para isso, basta criar as implementações para as abstrações dos objetos que serão feitos e uma nova **Abstract Factory** que os retorne. Por outro lado, esse padrão dificulta a criação de um novo tipo de objeto pertencente a essa família. Isso exigiria a adição de um método de criação na **Abstract Factory**, exigindo modificações em todas as suas implementações. Em resumo, é fácil adicionar uma nova família de objetos, porém é difícil adicionar um novo tipo de objeto na família.

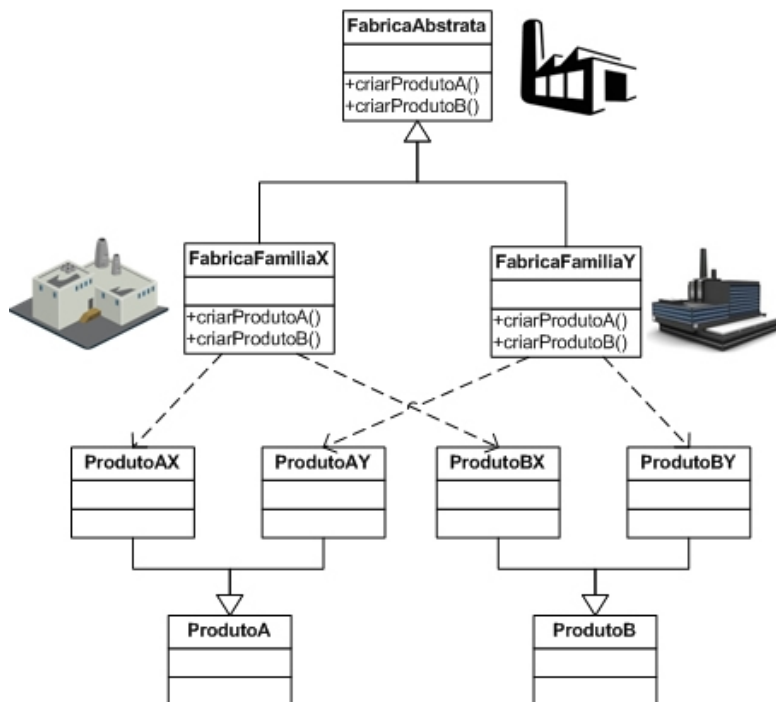



Figura 6.2: Estrutura Básica do Abstract Factory

Criando Objetos para o Envio de SMS

Vamos imaginar que uma aplicação precise interagir com o sistema de SMS de diversas operadoras de telefonia móvel. Também vamos supor que existam diversos objetos que precisem ser criados para interagir com esse serviço. Eles são:

- `ObservadorSMS`: Recebe eventos relativos à chegada de mensagens destinadas à aplicação.
- `FiltroSMS`: Configura um filtro que restringe as mensagens que devem ser recebidas ou tratadas por um determinado objeto.
- `EnviadorSMS`: Responsável por enviar SMS para dispositivos móveis.
- `RespondedorAutomatico`: Configura respostas automáticas para determinados tipos de mensagem.

Dependendo da operadora, alguns serviços, como a resposta automática ou o filtro para o recebimento de mensagens, pode ser implementados diretamente no servidor ou pela API cliente. Por esse motivo, não é possível, por exemplo, utilizar um filtro de uma operadora para filtrar as mensagens recebidas por um observador de outra operadora. O mesmo vale para a classe `RespondendorAutomatico` que também pode ser criada a partir de um filtro.

Dessa forma, o padrão **Abstract Factory** pode ser utilizado para concentrar a criação desses objetos em uma única classe. Esse objeto não precisa ser necessariamente apenas destinado a isso, podendo representar algum conceito do sistema. Nesse caso, a classe que irá criar os objetos pode ser a representação de uma conexão do sistema com os servidores da operadora. A seguir está representada a listagem que representa a abstração dessa classe.

Listagem 6.11 - Exemplo de utilização do Builder com interface fluente:

```
public interface ConexaoOperadora{
    public FiltroSMS criarFiltro(String expressao);
    public EnviadorSMS criarEnviador();
    public ObservadorSMS criarObservador();
    public ObservadorSMS criarObservadorComFiltro(FiltroSMS f);
    public RespondendorAutomatico criarRespostaAutomatica(FiltroSMS f);
    //outros métodos referentes a conexão
}
```

Pode-se observar pela listagem que existem métodos de criação que recebem uma instância de `FiltroSMS` como parâmetro. Caso um objeto criado a partir de uma operadora diferente fosse passado como parâmetro, um erro seria lançado, por mais que a abstração fosse aceita pelo tipo do parâmetro. A concentração desses métodos de criação na mesma classe ajuda a evitar que esse tipo de mistura de objetos ocorra.

O fato da **Abstract Factory** ser definida a partir de uma abstração permite que os clientes possam interagir de forma transparente com diferentes operadoras. Essa abstração também permite que cada classe tenha suas particularidades, como por exemplo, os parâmetros que são necessários para que a conexão possa ser estabelecida.

6.6 Considerações Finais do Capítulo

Esse capítulo focou em padrões para a criação de objetos. O padrão **`Static Factory Method`** encapsula a criação de objetos permitindo que sejam retornadas instâncias de subclasses, instâncias já existentes, e que se tenha uma expressividade maior a partir de nomes mais significativos. Foi visto também o padrão **`Singleton`** como um caso especial do **`Static Factory Method`** para o cenário em que deve haver apenas uma instância de uma determinada classe.

Para casos mais complexos, em que diversos parâmetros podem ser utilizados para construção do objeto, o **`Builder`** é um padrão adequado. Também foi mostrado como ele pode ser útil quando outros padrões que deixam a estrutura dos objetos mais complicada, como **`Proxy`**, **`Template Method`** e **`Composite`**, são empregados nas classes que precisam ser criadas. Finalmente, foi abordado o **`Abstract Factory`**, que foca em cenários onde famílias de objetos relacionados precisam ser criadas.

Capítulo 7

Modularidade

"Nosso objetivo definitivo é a programação extensível. Por isso queremos dizer a construção de uma hierarquia de módulos, cada um adicionando nova funcionalidade ao sistema."
– Niklaus Wirth

No capítulo anterior foram apresentados diversos padrões para a criação de objetos. Esses padrões, de uma forma geral, desacoplam a classe cliente da sua criação. Isso permite que essa classe dependa apenas da sua abstração, possibilitando que novas implementações sejam criadas e incorporadas à classe responsável pela criação. Apesar disso, a cliente acaba dependendo indiretamente da que está sendo criada. Por exemplo, ela pode depender de um **`Builder`**, que por sua vez depende das implementações. O mesmo vale para os outros padrões!

Essa dependência, mesmo que indireta, impede que a classe cliente e as implementações sejam divididas em módulos independentes. No exemplo citado do **`Builder`**, a classe cliente precisa ter acesso a ele, o qual precisa ter acesso às implementações. Com essa estrutura não é possível, por exemplo, inserir dinamicamente

novas implementações, permitindo que elas sejam utilizadas pela cliente, pois para isso a classe responsável pela criação, como o **`Builder`**, também precisaria de ser modificada.

A modularidade é cada vez mais um requisito não-funcional importante nas aplicações. Porém modularidade não é apenas dividir o software em módulos que formam um único bloco, mas permitir que novos módulos possam ser criados e incorporados no software sem a necessidade de sua modificação. Um exemplo de modularidade é o IDE Eclipse, ao qual novos plugins podem ser facilmente e dinamicamente incorporados.

Para que esse tipo de modularidade possa ser atingida, é necessário não somente a utilização de interfaces e abstrações para a interação entre os objetos, mas também um desacoplamento no momento de sua criação. Dessa forma, se uma classe cliente utilizar um **`Builder`** ou qualquer outro intermediário para instanciar seus objetos, ela não pode depender diretamente das implementações. Isso irá permitir que novas implementações sejam incorporadas sem modificar a aplicação.

7.1 Fábrica Dinâmica de Objetos

"Vidas fortes são motivadas por fins dinâmicos; as menores por desejos e inclinações."
– Kenneth Hildebrand

A primeira pergunta a ser respondida é: como obter um objeto sem depender direta ou indiretamente de sua classe? Em Java, a API de reflexão permite que uma classe seja instanciada a partir de uma string com o seu nome. Isso permite que as classes que serão instanciadas possam ser definidas, por exemplo, em um arquivo de configuração. Dessa forma, ao ler esse arquivo, é possível instanciar a classe sem depender diretamente dela. Isso permite que novas classes possam ser simplesmente adicionadas ao *classpath* da aplicação e configuradas no arquivo.

O padrão **`Dynamic Factory`** [17] propõe essa estrutura para a criação de objetos. Ela é aplicável quando se deseja criar um objeto de uma abstração conhecida, ou seja, que possui uma determinada interface ou superclasse, porém cuja implementação não pode ser determinada em tempo de compilação. Isso normalmente ocorre quando deseja-se permitir que novas implementações possam ser incorporadas ao software sem a sua modificação, tornando-o mais extensível e flexível. Um exemplo seria um software que precisa ser implantado em diversos clientes e em alguns deles é necessário desenvolver classes específicas para aquele contexto.

Estrutura do Padrão

A estrutura do padrão **`Dynamic factory`** não é muito diferente da estrutura de outros padrões de criação. Dessa forma, a classe cliente irá depender apenas da abstração compartilhada pelas implementações e irá acessar uma classe que implementa uma fábrica dinâmica, responsável pela criação dos objetos. Essa fábrica por sua vez acessa uma outra classe responsável por ler as informações a respeito de que implementação deve ser instanciada para uma determinada interface ou superclasse. A partir dessa informação, a API de reflexão é utilizada para instanciar essa classe.

A Figura 7.1 mostra a estrutura do padrão. A classe representada como `LeitorMetadados` é responsável por ler as informações a respeito de quais implementações devem ser instanciadas. Essas informações podem estar armazenadas em um arquivo de configuração, em um banco de dados ou até mesmo nas anotações de alguma classe. Independente da forma como essas configurações são definidas, é importante que elas contenham qual a implementação que deve ser instanciada para uma determinada abstração.

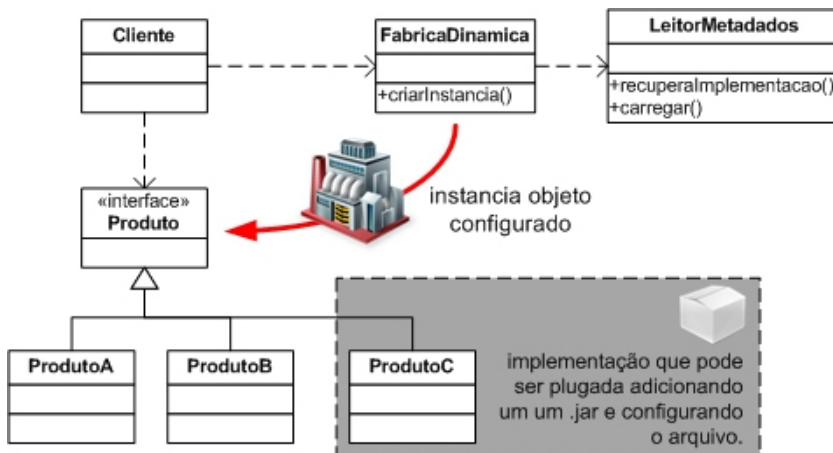



Figura 7.1: Estrutura do Dynamic Factory

A partir dessa estrutura é muito fácil introduzir no sistema novas classes. Para isso basta configurar o nome dessa classe no local onde o `LeitorMetadados` obtém as informações e adicionar a classe em um arquivo .jar no classpath da aplicação. Sendo assim, não apenas a classe `Cliente` irá depender apenas da abstração de `Produto`, mas o processo de criação dentro da classe `FabricaDinamica` também.

O QUE SÃO METADADOS?

A palavra "metadado" é sobrecarregada de significado dependendo da área da computação em que estamos. De uma forma geral, metadado significa "dado sobre o dado", que no contexto de uma linguagem orientada a objetos seriam as informações a respeito das classes de um sistema. Nesse caso, os dados são as informações da aplicação e as classes contêm informações sobre os seus tipos, sua visibilidade e como são manipulados. Dessa forma, no contexto de uma classe, os metadados seriam seus atributos, seus métodos, suas interfaces, sua superclasse, entre outros.

Na estrutura do padrão **`Dynamic Factory`**, a classe `LeitorMetadados` é responsável por ler uma informação a respeito de uma abstração, que é a classe que deve ser instanciada naquele contexto. Dessa forma, como essa é uma informação referente a uma classe do software, pode-se dizer que ele está lendo um novo metadado para ela.

O padrão **`Dynamic Factory`** pode ser implementado em conjunto com um **`Builder`** ou com um **`Static Factory Method`** de acordo com as necessidades da aplicação. Se necessário, mais de uma classe pode ser configurada para uma mesma abstração, assim como classes que serão utilizadas para composição, ou mesmo para envolver a classe principal, como um **`Proxy`**. O que diferencia esse padrão dos vistos no capítulo anterior é o desacoplamento entre a fábrica e as implementações que estão sendo criadas.

Dynamic Factory na Prática

Para exemplificar a criação de uma fábrica dinâmica com o uso de reflexão, a listagem a seguir mostra o exemplo de uma classe que cria os objetos de acordo com o nome da classe configurado em um arquivo de propriedades. A classe `FabricaDinamica` possui um construtor que recebe o nome do arquivo com as configurações de criação. A classe Properties da própria API padrão do Java é utilizada para fazer essa leitura, fazendo o papel de leitora de metadados.

Listagem 7.1 - Fabrica dinâmica que lê a classe da implementação de um arquivo de propriedades:

```
public class FabricaDinamica {

    private Properties props;

    public FabricaDinamica(String arquivo)
                throws FileNotFoundException, IOException{
        props = new Properties();
        props.load(new FileInputStream(arquivo));
    }

    public <E> E criaImplementacao(Class<E> interface) {
        String nomeClasse = props.getProperty(interface.getName());

        if(nomeClasse == null) {
            throw new IllegalArgumentException(
                "Interface não configurada");
        }

        try {
            Class clazz = Class.forName(nomeClasse);
            if (interface.isAssignableFrom(clazz)) {
                return (E) clazz.newInstance();
            } else {
                throw new IllegalArgumentException(
                    "Classe configurada não implementa a interface");
            }
        } catch (ClassNotFoundException e) {
            throw new IllegalArgumentException(
                "Classe configurada não existe",e);
        } catch (Exception e) {
            throw new IllegalArgumentException(
                "Não foi possível criar a implementação",e);
        }
    }

}
```

Muitas vezes, classes já existentes no sistema ou em algum framework utilizado pela aplicação podem ser utilizados para ser um participante de um padrão. No exemplo apresentado da **`Dynamic factory`**, foi utilizada a classe `Properties` para realização da leitura do arquivo com os metadados. Em outros padrões, como

o **Observer**, por exemplos, o papel de observador ou observado pode ser desempenhado por alguma classe já existente. Caso seja requerida uma interface diferente para a classe existente, o padrão **Adapter** pode ser utilizada para fazer a adaptação.

A classe do exemplo espera que no arquivo sejam definidas propriedades com o nome da interface e valores com o nome da classe que deve ser instanciada. Dessa forma, quando o método `criaImplementacao()` é invocado, o nome da interface é utilizado para recuperar a respectiva implementação que deve ser instanciada. Essa instanciação é feita, primeiramente obtendo-se a instância de `Class` referente àquela implementação através do método estático `Class.forName()`, e em seguida invocando o método `newInstance()`. Observe que antes de efetivamente criar o objeto é verificado se a classe realmente implementa a interface. Nesse caso, com a chamada do método `newInstance()`, o construtor sem parâmetros da classe será invocado. Sendo assim, para que essa fábrica seja utilizada, é preciso que as implementações possuam esse construtor.

Reflexão em Java

Reflexão é a capacidade de um sistema de executar computações a respeito de si mesmo. De uma forma mais prática, isso envolve o sistema poder obter informações, acessar e até mesmo modificar suas próprias classes. A API Reflection da linguagem Java provê funcionalidades apenas para obter informações e acessar as classes do sistema. Funcionalidades de modificação de classes são mais comuns em linguagens dinâmicas.

A classe básica da API Reflection é a `Class` que representa uma classe do sistema. A partir dela é possível obter informações, como seu nome, seus métodos, seus atributos, sua superclasse e suas interfaces. Além disso, também é possível instanciá-la através do método `newInstance()`, como visto no exemplo. Outras classes dessa API representam outros elementos de código, como `Method`, `Field` e `Constructor`. Está fora do escopo desse livro a explicação do funcionamento dessa API, porém é importante saber esses fundamentos para a compreensão de como o padrão **Dynamic Factory** funciona.

Para exemplificar a utilização da classe `FabricaDinamica`, será retomado o exemplo do gerador de arquivos. Para simplificar o exemplo, vamos supor que as implementações possuem um construtor sem parâmetros e que o pós-

processador é configurado através de um método setter. A versão apresentada da classe `FabricaDinamica` não suporta o carregamento de mais de uma implementação da mesma interface, porém isso poderia ser contornado com uma pequena modificação. Para manter o exemplo simples, será configurada apenas uma instância da interface `PosProcessador` para ser inserida no `GeradorArquivo`.

A seguir são apresentadas listagens que mostram um exemplo de como o arquivo de propriedades seria definido e como a criação das classes seria realizada. Observe que o código que instancia o `GeradorArquivo` com sua dependência não referencia em nenhum momento a implementação. Adicionalmente, a implementação também não é referenciada pela classe `FabricaDinamica`, que lê o nome da classe de um arquivo externo. Esse tipo de prática permite que novas classes sejam criadas e plugadas no sistema, sem a necessidade da recompilação das classes já existentes.

Listagem 7.2 - Arquivo de propriedades com a definição das classes:

```
br.com.casadocodigo.GeradorArquivo=br.com.casadocodigo.GeradorXML
br.com.casadocodigo.PosProcessador=br.com.casadocodigo.Compactador
```

Listagem 7.3 - Utilizando a FabricaDinamica para a criação do GeradorArquivo:

```
FabricaDinamica f = new FabricaDinamica("configuracoes.prop");
GeradorArquivo gerador = f.criaImplementacao(GeradorArquivo.class);
gerador.setPosProcessador(f.criaImplementacao(PosProcessador.class);)
```

Uma Base Para Outros Padrões

Mesmo sendo um padrão importante, a **`Dynamic Factory`** é muitas vezes utilizada por baixo dos panos. Apesar de ser válida sua utilização direta para a criação de alguns objetos do sistema, pode ser complicado utilizar esse padrão como estratégia de criação geral dentro de uma arquitetura. As próximas seções apresentam padrões para a criação desacoplada de objetos que podem ser utilizados no lugar da **`Dynamic Factory`**. Observem que, para que eles sejam possíveis de ser implementados para se obter modularidade, ainda é necessário que uma **`Dynamic Factory`** atue no carregamento dinâmico dessas classes em algum momento.

7.2 Injeção de Dependências

"A verdadeira felicidade é... aproveitar o presente, sem a ansiedade da dependência do futuro."
– Lucius Annaeus Seneca

Quando um objeto precisa da colaboração de outro para cumprir suas responsabilidades, é um processo natural que o próprio objeto crie ou busque essas instâncias. Essa responsabilidade pela criação das dependências, quando está no próprio objeto, pode acabar criando um acoplamento e prejudicando a modularidade, mesmo quando interfaces são utilizadas para interação entre as classes.

Uma forma de evitar que a própria classe seja responsável por criar suas dependências é criá-las de forma externa e inseri-las no objeto no momento ou depois de sua criação. Essa é a ideia principal do padrão **`Dependency Injection`** [13]! Apesar de se utilizar muito o nome em português, **`Injeção de Dependências`**, para padronizar a utilização dos nomes dos padrões em inglês nesse livro, ele será referenciado como **`Dependency Injection`**.

Inversão de Controle?

Devido ao fato desse padrão inverter a responsabilidade de criação das dependências de uma classe, ele também é conhecido como Inversão de Controle. Porém o nome "inversão de controle" também é utilizado para denominar práticas para o desenvolvimento de frameworks, nas quais as classes do framework invocam classes da aplicação, invertendo assim o controle do fluxo de execução. Dessa forma, o nome **`Dependency Injection`** acabou prevalecendo.

Apesar de sua simplicidade, o **`Dependency Injection`** é um padrão muito poderoso, pois desacopla a classe de suas dependências, permitindo que elas sejam reutilizadas em diferentes contextos. Consequentemente, a testabilidade dessa classe acaba também sendo aumentada, pois objetos voltados para o teste podem ser injetados no lugar das dependências.

Funcionamento da Injeção

O foco do padrão **Dependency Injection** é na criação e configuração de uma classe que possui uma outra como dependência. Essa dependência, por sua vez, é do tipo de uma abstração, normalmente representada por uma interface, a qual geralmente possui mais de uma implementação. O problema no caso é como criar e conectar essas instâncias sem que exista uma dependência de uma para outra. Esse padrão propõe como solução a existência de uma outra classe que irá "montar" a aplicação, criando as implementações corretas de cada componente e conectando-as de forma a se obter o comportamento desejado.

A Figura 7.2 apresenta uma representação do padrão **Dependency Injection**. A classe `Montador` é quem na verdade coordena todo o processo, tanto de criar as instâncias corretas, quanto de injetá-las na classe `Produto`. Essa, por sua vez, deve apenas prover uma forma de as dependências poderem ser injetadas. A próxima seção explora mais detalhes a respeito desse processo.

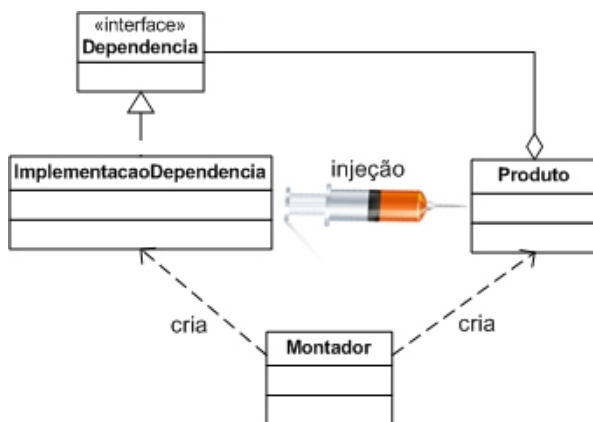



Figura 7.2: Injeção de Dependências

O processo de criação e reutilização de instâncias é controlado pelo `Montador`, que pode definir diferentes estratégias para isso. Por exemplo, caso a dependência seja um objeto que possa ser compartilhado, a mesma instância pode ser injetada diversas vezes. Uma outra estratégia, muito comum para conexões a serviços ou servidores remotos, é a criação de um **pool** de objetos, de onde eles são retirados para a injeção em uma classe e retornados ao final de sua utilização. O fato desse processo não ser controlado pela classe permite que ela possa ser reutilizada em diversos con-

textos.

Imagine uma classe que realize acesso a uma base de dados e receba através de uma `Dependency Injection` a conexão que deve ser utilizada por ela. Ela poderia ser utilizada por uma aplicação desktop onde apenas uma conexão é criada para toda aplicação, mas também poderia ser reusada em uma aplicação Java EE que executa em um servidor de aplicação no qual a conexão é obtida de um **pool**. O simples fato dessa conexão ser criada de forma externa à classe e injetada em sua estrutura é que a torna reutilizável nas diferentes arquiteturas citadas.

Como pode-se perceber, grande parte do trabalho acaba ficando com a classe `Montador`, responsável pela criação e conexão entre as classes envolvidas. Ela pode ser simples e criada especificamente para uma aplicação, ou ser genérica e carregar dinamicamente as classes que precisam ser instanciadas através de uma `Dynamic Factory`. Felizmente, existem frameworks e APIs, como o Spring e o CDI do Java EE, que implementam montadores que criam as instâncias a partir de configurações em arquivos XML e anotações. Sendo assim, raramente é necessário criar uma classe para fazer o papel de montador. Mais à frente será mostrado como a injeção de dependências funciona no Spring.

Formas de Injetar Dependências

A injeção de dependências pode ser habilitada de diversas formas em uma classe. Essa seção irá explorar as alternativas existentes, assim como a motivação na utilização de cada uma delas. De alguma forma, deve ser possível que uma entidade externa reconheça a dependência e possa inseri-la na classe, mesmo que em apenas certos momentos de seu ciclo de vida.

O tipo mais comum de injeção de dependências é através do método **setter**. Através desse método, a classe permite que a dependência seja configurada com um método **setter**, que a recebe como parâmetro. A listagem a seguir mostra como a classe `AcessoDados` disponibiliza um método **setter** para que a conexão com a base de dados possa ser configurada. Utilizando essa estratégia, a dependência pode ser atribuída e alterada a qualquer momento.

Listagem 7.4 - Injeção de dependência por método setter:

```
public class AcessoDados {
    private Connection connection;
    public void setConnection(Connection c) {
        connection = c;
```

```
    }
    //...
}
```

Uma outra abordagem para a injeção de dependências é o construtor. Nessa estratégia, a dependência deve ser injetada no objeto no momento de sua criação. Esse caso acaba sendo mais indicado para cenários onde ela é obrigatória e essencial para o funcionamento da classe. A listagem a seguir mostra o mesmo exemplo da classe `AcessoDados`, porém utilizando a **`Dependency Injection`** através do construtor.

Listagem 7.5 - Injeção de dependência pelo construtor:

```
public class AcessoDados {
    private Connection connection;
    public AcessoDados(Connection c) {
        connection = c;
    }
    //...
}
```

Como foi apresentado no capítulo anterior, a utilização de construtores traz diversas restrições e a classe que utiliza injeção por construtores acaba sofrendo os mesmos problemas. Por outro lado, o construtor é uma forma de garantir que uma instância da classe não será criada sem receber aquele parâmetro. Uma dependência bidirecional, por exemplo, seria impossível de ser injetada por construtores nas duas classes... Na prática, a injeção por métodos **setter** acaba sendo mais comum, sendo feita uma configuração cuidadosa para que o montador não deixe de configurar as dependências importantes.

Uma outra abordagem existente para a **`Dependency Injection`** é a utilização de interfaces. Nesse caso, é definida uma interface que declara métodos que devem ser utilizados para a injeção da dependência. Dessa forma, a classe que precisa dela deve implementar essa interface para poder recebê-la. Apesar de ser uma abordagem mais "burocrática" que as outras duas, ela permite que a classe que injeta a dependência reconheça facilmente o objeto que precisa dela para poder proceder com a injeção.

A estratégia de utilização de interfaces para injetar dependências é muito utilizada quando o montador precisa gerenciar diversas instâncias e passar certos objetos apenas para as classes que precisam deles. A interface acaba funcionando como um marcador que indica quando o objeto precisa da dependência.

Para exemplificar essa situação, considere um sistema que possui uma abstração chamada `Processo` que representam processos de negócio que podem ser executados. Das classes que implementam esses processos, algumas delas precisam saber qual é o usuário que as está executando. Para não tornar essas classe acopladas a como as instâncias da classe `Usuario` são armazenadas e acessadas, tomou-se a decisão de utilizar **Dependency Injection** para que o usuário seja inserido nessas instâncias. A listagem a seguir mostra a definição da interface que as classes que desejam receber a instância de **Usuario** devem implementar.

Listagem 7.6 - Definição de um interface para injeção do usuário:

```
public interface CienteUsuario{
    public void recebeUsuario(Usuario u);
}
```

Como forma de injetar a dependência, foi criado um **Proxy** que encapsula classes da interface `Executor`, que é responsável por executar as classes do tipo `Processo` passadas a elas. A classe `ProxyInjecaoUsuario`, apresentada na próxima listagem, verifica com o operador `instanceof` se o objeto implementa a interface e injeta a dependência da classe `Usuario` em caso positivo.

Listagem 7.7 - Proxy para a injeção do usuário:

```
public class ProxyInjecaoUsuario implements Executor {
    private Executor executor;
    private Usuario usuario;
    public ProxyInjecaoUsuario(Executor e, Usuario u) {
        executor = e;
        usuario = u;
    }
    public void executar(Processo p) {
        if(p instanceof CienteUsuario) {
            ((CienteUsuario)p).recebeUsuario(usuario);
        }
        executor.executar(p);
    }
}
```

Pelas técnicas anteriores de injeção de dependências, pode ser observado que a classe deve sinalizar de alguma forma onde e qual ela que precisa ser injetada. Uma outra forma de se sinalizar isso, bastante utilizada em frameworks e APIs

mais recentes, é através de anotações. A anotação é normalmente incluída em um atributo ou no seu respectivo método **setter** no qual ela deve ser injetada. A listagem a seguir mostra um exemplo de um Servlet que deve receber uma injeção de dependência de um EJB do servidor de aplicações onde está implantado. Note que nesse caso é feita utilização de reflexão para que a dependência possa ser inserida diretamente em um atributo, mesmo ele sendo privado.

Listagem 7.8 - Injeção de dependência pelo construtor:

```
public class ServletLogin extends HttpServlet {
    @EJB
    private ServicoAutenticacao servico;
    //lógica do servlet omitida
}
```

Os Efeitos Colaterais da Injeção

Se por um lado a **`Dependency Injection`** tem o potencial de tornar a classe mais reutilizável, a passagem de responsabilidade da montagem do relacionamento das classes para um terceiro também pode gerar erros em tempo de execução. Como a classe não tem mais controle sobre suas dependências, ela fica sujeita a erros referentes a uma falha nesse processo que ocorre de forma externa.

Um erro bastante comum quando esse padrão é utilizado ocorre quando uma dependência deixa de ser configurada. Nesse caso, é lançada a conhecida e indesejável `NullPointerException` quando a classe tenta invocar algum método nela. Por uma questão de segurança de código, isso pode levar muitos desenvolvedores a incluírem diversos condicionais para verificar se a instância não é nula, o que gera uma certa poluição e verbosidade.

Apesar da injeção de dependências simplificar os testes de unidade, devido à chance de erro com as montagens, quando esse padrão é utilizado, é muito importante que se faça os testes de integração da aplicação. Esses testes servem para garantir não somente que as classes estão funcionando de forma adequada juntas, mas que a aplicação está criando e injetando as instâncias de forma correta. Seria como se estivéssemos testando se o montador está fazendo a injeção adequadamente.

A montagem dinâmica da aplicação também pode dificultar a compreensão do contexto global de como os objetos interagem para formar a funcionalidade da aplicação. A flexibilidade de uma classe em poder interagir com diversas implementações também dificulta saber com que classes ela está interagindo em um contexto

concreto. Por exemplo, ao se deparar com um erro, o desenvolvedor não tem como saber qual foi a classe invocada por ela. Para isso ele deve procurar no montador ou nas configurações da aplicação para saber qual era a estrutura naquele caso. Essa indireção acaba sendo um efeito colateral ao desacoplamento conquistado.

Injeção de Dependências no Spring

Normalmente, quando a injeção de dependências é utilizada em uma aplicação, é utilizado algum framework para realizar a montagem das classes. Um dos frameworks que primeiro se destacou por esse tipo de funcionalidade e muito utilizado até hoje é o Spring. Utilizando-o, as instâncias com suas respectivas dependências são definidas em um arquivo XML e o framework se encarrega de criá-las.

A listagem a seguir exemplifica como o Spring seria utilizado no exemplo do gerador de arquivo. Cada instância é definida como um **bean** no arquivo de configuração, recebendo um nome que será utilizado para referências internas e para recuperá-lo de forma externa. Utilizando o nome dos **beans**, as instâncias podem ser injetadas utilizando diferentes estratégias, como pelo construtor ou através de propriedades.

Listagem 7.9 - Configuração da injeção de dependência pelo Spring:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans...>
   <bean id="compactador" class="br.com.casadocodigo.Compactador" />
   <bean id="criptografador" class="br.com.casadocodigo.Criptografador">
           <constructor-arg type="int" value="3"/>
   </bean>
   <bean id="composite"
           class="br.com.casadocodigo.PosProcessadorComposto">
           <constructor-arg>
            <ref bean="compactador"/>
            <ref bean="criptografador"/>
        </constructor-arg>
   </bean>
    <bean id="geradorArquivo" class="br.com.casadocodigo.GeradorXML">
          <property name="processador" ref="composite" />
   </bean>
</beans>
```

No exemplo, o **bean** `criptografador` recebe um parâmetro no constru-

tor referente ao tamanho do deslocamento de caracteres utilizado no algoritmo. As referências dos **beans** `criptografador` e `compactador` são também passadas como parâmetro para o construtor do **bean** `composite`. Finalmente, o **bean** `geradorArquivo` recebe uma injeção do **bean** `composite` na propriedade `processador`. Através do arquivo XML apresentado, é possível definir qual classe será instanciada para cada **bean** e como uma deve ser utilizada na construção da outra.

Os usuários do framework Spring, para recuperarem um **bean** devem fazer uso da classe `ApplicationContext`. Ela é responsável por carregar o arquivo XML com a definição da composição dos objetos e retorná-los à aplicação. A listagem a seguir mostra como ela seria utilizada na criação de uma instância do `GeradorArquivo` como definido no arquivo XML. Em aplicações web, o Spring normalmente é integrado diretamente com o container web já criando os objetos e injetando-os nas classes que tratam as requisições. Nesse caso, não há a necessidade de acesso direto à classe `ApplicationContext` pela aplicação, pois isso é feito pelo container web ou pelo framework que está sendo utilizado.

Listagem 7.10 - Recuperação de um bean configurado no Spring:

```
String xml = "applicationContext.xml";
ApplicationContext ac = new ClassPathXmlApplicationContext(xml);
GeradorArquivo ga = (GeradorArquivo) ac.getBean("geradorArquivo");
```

A partir do que foi mostrado, é possível perceber como seria simples a mudança das implementações envolvidas e da configuração da estrutura montada. Por serem definidos em um arquivo XML, os **beans** podem ser alterados, consequentemente mudando o comportamento da aplicação, sem a necessidade de sua recompilação. Para exemplificar esse fato, imagine que seja necessário introduzir um **`Proxy`** na classe `GeradorArquivo` para a invocação assíncrona de seus métodos. Isso poderia ser facilmente realizado colocando a classe do **`Proxy`** no **bean** `geradorArquivo`, fazendo-o encapsular o que estava definido anteriormente, conforme mostrado na listagem a seguir.

Listagem 7.11 - Inserção de um Proxy de forma transparente:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans...>
   ...
   <bean id="geradorArquivo"
```

```
            class="br.com.casadocodigo.GeradorProxyAssincrono">
          <constructor-arg>
            <ref bean="geradorEncapsulado"/>
        </constructor-arg>
   </bean>
   <bean id="geradorEncapsulado" class="br.com.casadocodigo.GeradorXML">
          <property name="processador" ref="composite" />
   </bean>
</beans>
```

7.3 Service Locator

"Quando tentar localizar algo, procure primeiro em sua mente."
– Craig Bruce

O padrão **Service Locator** [13] é uma alternativa ao padrão **Dependency Injection** para permitir a modularidade nas aplicações. Diferentemente da injeção, as próprias classes são responsáveis por buscar as suas dependências. Porém, elas fazem isso através de uma outra classe responsável por localizar a implementação que deve ser utilizada para uma determinada interface.

No contexto desse padrão, a palavra "serviço" é utilizada para denotar uma colaboração de uma outra classe. Seria como se a classe externa estivesse prestando um serviço para a que a utiliza. Nesse caso, a classe principal declara o tipo de serviço que ela precisa utilizando uma abstração, como uma classe abstrata ou uma interface, e então utiliza um **Service Locator** para encontrar a que irá prestar aquele serviço para ela.

Service Locator como um J2EE Pattern

O **`Service Locator`** foi documentado como um Core J2EE Pattern e foi muito utilizado para a localização de serviços remotos, principalmente enterprise beans, em aplicações J2EE [6]. Nesse contexto, o **`Service Locator`** se conectava a um repositório JNDI para adquirir uma referência a esses serviços remotos e fazia um cache dessa referência para evitar acessos desnecessários ao registro de nomes. Nas novas versões da plataforma enterprise Java, que perdeu o "2" e agora se chama simplesmente Java EE, a utilização desse padrão ficou obsoleta a própria plataforma já presta esse serviço, realizando inclusive a injeção dessas dependências nas classes.

Por esse motivo, quando se fala em **`Service Locator`** muita gente acha que é um padrão obsoleto pela experiência que teve com ele na plataforma J2EE. Nesse contexto, esse padrão tinha o objetivo principal de centralizar o mecanismo de busca de serviços, encapsulando as peculiaridades de cada registro e evitando buscas desnecessárias. Porém, fora desse contexto, esse padrão ainda é muito importante, principalmente para permitir a modularidade das aplicações. Sendo assim, se você ainda guarda algum trauma da utilização desse padrão para buscar EJBs, deixe isso de lado porque ele pode ser muito útil em outras situações.

Estrutura para Localização de Serviços

No **`Service Locator`**, não existe a figura de uma classe que centraliza a responsabilidade de criação e ligação das classes que irão compor a aplicação. Nesse padrão cada uma busca suas próprias dependências através de uma classe responsável por buscar a instância que deve ser adicionada como dependência.

A Figura 7.3 apresenta a estrutura do padrão **`Service Locator`**. No diagrama, a classe `Cliente` representa a classe que precisa criar uma instância da abstração `Servico`. Apesar de haver uma relação de agregação entre essas classes, nesse padrão a relação poderia ser mais fraca, como a criação e utilização do serviço de forma encapsulada dentro de um método. Segundo o padrão, uma classe responsável por encontrar e retornar a implementação desse serviço, representada por `LocalizadorServicos`, é utilizada pela classe `Cliente` para fazer isso.

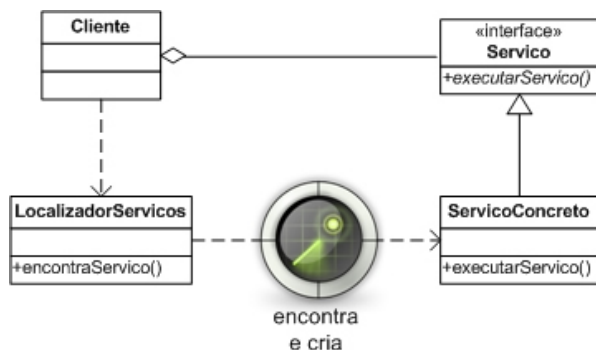



Figura 7.3: Estrutura do Service Locator

Uma classe que implemente um localizador de serviços pode possuir uma implementação fixa, retornando as instâncias desejadas de acordo com o parâmetro passado. Era esse tipo de **`Service Locator`** que costumava ser utilizado nas aplicações J2EE. Porém, para conseguir modularizar a aplicação, é preciso que ele faça uso de uma **`Dynamic Factory`** para que os serviços retornados possam ser mais facilmente alterados.

Quando o padrão **`Dependency Injection`** é utilizado, essa questão ainda precisa ser implementada, porém quem é o responsável por buscar os serviços não é alguém que é invocado pelas classes, e sim o montador. Por isso, muitas vezes, o montador precisa de um **`Service Locator`** para poder localizar as implementações e injetá-las nas classes.

Um dos problemas desse padrão está no fato das classes ficarem acopladas à classe responsável pela busca de serviços. Isso as impede de serem reutilizadas fora de um contexto em que o **`Service Locator`** esteja presente. De qualquer forma, o **`Service Locator`** costuma ser utilizado encapsulado na classe, não sendo uma dependência exposta na API externa. Isso é bom pois não cria essa dependência nas classes cliente.

Utilizando a classe ServiceLoader

A JDK possui uma classe chamada `ServiceLoader` que pode ser utilizada como um **`Service Locator`** que carrega dinamicamente as implementações de uma interface. Essas implementações podem estar empacotadas e configuradas dentro de diferentes arquivos JAR. Sendo assim, a implementação será carregada de

acordo com os arquivos JAR que estiverem presentes no classpath da aplicação. Basta incluir e remover arquivos do JAR para alterar a implementação utilizada.

Para que a classe `ServiceLoader` possa ser utilizada, o primeiro passo seria ter uma interface que serviria de abstração para o serviço a ser criado. As implementações dessa interface poderiam então ser colocadas em arquivos JAR para serem localizadas. Além da implementação, ainda é necessário incluir no JAR um arquivo que registra a classe criada como uma implementação daquele serviço. Esse arquivo deve ser colocado dentro do diretório `META_INF/services` e deve ter o nome completo da interface, incluindo o pacote. Como conteúdo, deve haver simplesmente uma linha com o nome da classe para cada implementação daquela abstração contida naquele JAR.

A Figura 7.4 ilustra como deve ser essa estrutura. Considere `Abstracao` como a interface para qual se deseje definir os serviços. No exemplo, o `arquivo1.jar` possui as implementações `ImplementacaoA` e `ImplementacaoB` e o `arquivo2.jar` possui a `ImplementacaoC`. Cada arquivo JAR possui um arquivo chamado `Abstracao`, o nome da interface, com os nome das classes. Dessa forma, de acordo com o arquivo JAR colocado no classpath da aplicação, classes diferentes seriam retornadas pelo `ServiceLoader`.

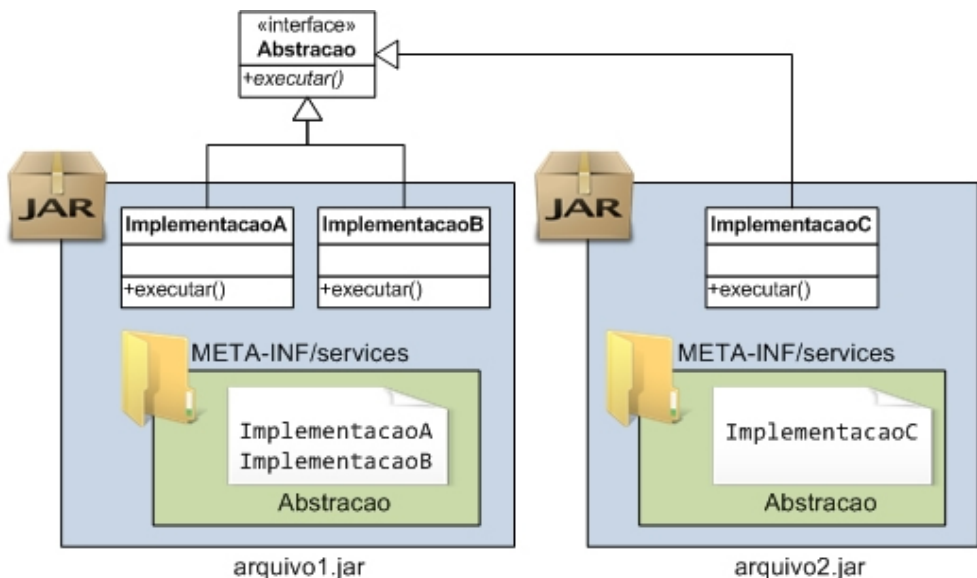



Figura 7.4: Arquivos JAR para utilização do ServiceLoader

A listagem a seguir ilustra, dentro do exemplo da Figura 7.4, como funciona a classe `ServiceLoader`. Esse trecho de código imprime no console todas as implementações que estão registradas para uma interface. O método estático `load()` da classe `ServiceLoader` é utilizado para carregar todas as implementações registradas da interface `Abstracao`. Em seguida, é utilizado um `Iterator` para percorrer as implementações recuperadas e imprimi-las no console.

Listagem 7.12 - Imprimindo as implementações encontradas pelo ServiceLoader:

```
ServiceLoader<Abstracao> sl = ServiceLoader.load(Abstracao.class);
Iterator<Abstracao> i = sl.iterator();
while(i.hasNext()) {
    System.out.println(i.next().getClass().getName());
}
```

Para mostrar um exemplo mais concreto do uso do `ServiceLoader`, a listagem a seguir mostra como ele poderia ser utilizado para a recuperação dos pós-processadores para a classe `GeradorArquivo`. Nesse caso, o **Service Locator** está sendo utilizado no construtor para popular o atributo `processador` da classe. Observe que caso exista apenas uma implementação, ela será atribuída diretamente ao atributo, mas caso exista mais de uma, a classe `PosProcessadorComposto` é utilizada. Dessa forma, caso um arquivo JAR configurado com uma nova implementação for adicionado ao classpath da aplicação, ela será recuperada pelo `ServiceLoader`. No caso da classe `PosProcessadorComposto`, que é uma implementação que não deve ser retornada, basta não incluí-la no arquivo.

Listagem 7.13 - Buscando os pós-processadores do GeradorArquivo com um Service Locator:

```
public abstract class GeradorArquivo {

    private PosProcessador processador = new NullProcessador();

    public GeradorArquivo()) {
        ServiceLoader<PosProcessador> sl =
                ServiceLoader.load(PosProcessador.class);
        Iterator<PosProcessador> i = sl.iterator();
        List<PosProcessador> lista = new ArrayList<>();
```

```
        while(i.hasNext()) {
            lista.add(i);
        }
        if(lista.size() == 1) {
            processador = lista.get(0);
        } else if (lista.size > 1) {
            processador = new PosProcessadorComposto(lista);
        }
    }

    //demais métodos
}
```

Combinando diversos padrões em uma solução

No decorrer desse livro, vimos diversos padrões que aos poucos foram sendo incorporados às soluções apresentadas. A ideia é que paulatinamente eles comecem a fazer parte da nossa forma de pensar e que acabamos utilizando-os mesmo sem perceber. Desafio você a dar mais uma olhada no código da última listagem e tentar identificar os padrões que estão ali presentes. Utilizamos o **Bridge** na abstração do `GeradorArquivo` e no uso de composição para os pós-processadores, o **Null Object** é utilizado para os casos onde não há pós-processador, o **Composite** é utilizado para quando há mais de um pós-processador, e, finalmente, o **Service Locator** é utilizado para o carregamento dinâmico dos pós-processadores.

Note que não estou dizendo para aplicar todos os padrões possíveis em uma mesma solução, pois isso é uma má prática. Porém, o que acaba acontecendo, é que na solução sucessiva dos problemas de design de uma classe, é natural que o resultado seja fruto da combinação de diversos padrões.

7.4 Service Locator versus Dependency Injection

Depois de aprender sobre os padrões **Dependency Injection** e **Service Locator**, surge a dúvida de qual deles utilizar. Antes de começar a explorar as vantagens e desvantagens de cada um, é importante ressaltar que com a utilização de

qualquer um dos dois é possível se obter modularidade, particionando a aplicação em partes efetivamente independentes. Dessa forma, fica mais simples a introdução de novas implementações sem a necessidade de recompilação do código das classes que as utilizam. Com isso, cada um dos módulos do sistema pode evoluir de forma independente, facilitando a manutenção e evolução do software.

Uma das grandes diferenças entre os dois padrões está em quem possui a responsabilidade de montar a configuração de objetos da aplicação. Quando o padrão **`Dependency Injection`** é utilizado, ela fica centralizada na classe responsável pela montagem. Nesse caso, ela precisa conhecer todas as dependências de todas as classes envolvidas, estaticamente ou dinamicamente a partir de alguma configuração. Diferentemente, quando se utiliza o **`Service Locator`**, cada classe é responsável por buscar suas dependências a partir de uma classe responsável pela localização dos serviços. Isso permite que novas classes definam novos pontos onde outras abstrações possam ser inseridas, sem a necessidade de haver um ponto central que saiba dessa nova dependência.

Em resumo, quando se injeta dependências, a responsabilidade de montar toda a aplicação fica centralizada e, quando cada classe busca seus serviços, essa responsabilidade fica distribuída. Sendo assim, se o contexto é uma aplicação onde se deseja ter modularidade para questões de manutenção e evolução, mas não existe a necessidade da adição de novas classes dinamicamente, então o **`Dependency Injection`** é uma boa solução. Porém, em cenários em que a arquitetura é baseada em plugins e não existe um ponto na aplicação no qual se tem consciência de todas possíveis dependências, então o **`Service Locator`** se mostra mais adequado.

Uma consequência disso está em quão explícitas são as dependências de uma classe. Quando se usa o **`Dependency Injection`**, é fácil visualizar através dos construtores, interfaces e métodos de acesso quais são as dependências que podem ser injetadas. Por outro lado, a invocação de um **`Service Locator`** pode estar encapsulada dentro dos métodos da classe, não as expondo externamente. Isso pode ter um efeito negativo, pois não deixa claro quais precisam ser configuradas para uma determinada classe.

Apesar de não ser obrigatório, o **`Dependency Injection`** acaba sendo muito utilizado em ambientes onde os objetos da aplicação são gerenciados por um container. Um exemplo disso seriam aplicações web ou aplicações EJB, em que o ciclo de vida dos objetos é controlado pelo servidor. Dessa forma, como a criação dos objetos que tratarão as requisições fica a cargo do container, ele tem a oportunidade de injetar as dependências no momento certo. Frameworks como Spring são chamados

de *lightweight container*, ou container leve, devido ao fato de controlarem o ciclo de vida dos objetos gerenciados por eles.

Um ponto onde o **`Dependency Injection`** certamente leva vantagem é na testabilidade. Como a dependência é injetada na classe, esse mesmo mecanismo pode ser utilizado para a injeção de um objeto falso, ou *mock object* [25], para isolar a classe durante o teste. Já com o **`Service Locator`**, isso pode ser um pouco mais complicado, pois o *mock object* precisaria ser configurado para ser retornado por ele. Dependendo de como essa configuração é feita, isso pode ser mais simples ou mais complicado, porém de qualquer forma pode ser um problema para casos em que a classe do *mock* é gerada dinamicamente por algum framework. Nesses casos, uma solução seria já pensar em uma implementação que simplifique a substituição da implementação retornada para um serviço em código de teste.

Alternativas de Projeto

Os padrões de projeto documentam soluções recorrentes que são utilizadas em um determinado contexto. Por mais que eles documentem boas soluções, nem sempre um determinado padrão é a melhor solução para uma aplicação. Por isso, podem existir mais de um padrão que podem ser aplicados na resolução de um mesmo problema, como mostrado nesse capítulo. Se você quer flexibilizar os passos de um algoritmo, você pode utilizar o **`Template Method`** para fazer isso com herança, ou o **`Strategy`** para usar composição. Se você quer combinar soluções de classes existentes de forma transparente, pode-se utilizar o **`Composite`** ou o **`Chain of Responsibility`**. Por mais que ambos os padrões nesses casos resolvam o problema, eles possuem características e consequências diferentes. Dessa forma, é importante não utilizar os padrões cegamente e avaliar dentre as alternativas existentes quais as que possuem consequências mais adequadas para o contexto da sua aplicação.

7.5 Considerações Finais do Capítulo

Nesse capítulo abordamos padrões que focam em soluções para que se consiga obter modularidade para aplicações. Essas soluções atuam em um momento crucial na

vida dos objetos: a sua criação. Desde o primeiro capítulo desse livro, já foram apresentadas soluções que utilizam encapsulamento e polimorfismo para desacoplar os objetos, porém no momento da criação, em que a implementação efetivamente precisa ser instanciada, a referência direta à classe costuma interferir na modularidade.

O padrão **`Dynamic Factory`** foi apresentado como uma base que pode ser utilizada pelos outros padrões para o carregamento dinâmico das implementações que devem ser utilizadas para compor a classe principal. Em seguida, foram msotrados os padrões **`Dependency Injection`** e **`Service Locator`**, que apresentam alternativas a respeito de como uma classe pode delegar a criação de suas dependências. Se combinados com uma solução que utiliza um **`Dynamic Factory`** na criação das instâncias, é possível realmente modularizar a aplicação.

Capítulo 8

Adicionando Operações

"Nós não podemos resolver problemas usando o mesmo tipo de pensamento que usamos quando nós os criamos."
– Albert Einstein

Nos capítulos anteriores desse livro foram apresentados alguns padrões que auxiliam na adição de funcionalidade em classes já existentes. Por exemplo, os padrões **`Proxy`** e **`Decorator`** permitem a adição de funcionalidades através do encapsulamento do objeto original. Dessa forma, um **`Proxy`** pode adicionar uma lógica antes ou depois da invocação de um método. Porém essa lógica adicionada é considerada transversal à da classe que está sendo encapsulada, ou seja, ela é independente e complementar a elas.

Por outro lado, padrões que utilizam composição, como o **`Strategy`** e o **`Bridge`**, podem trocar a instância que a compõe para alterar a funcionalidade que está sendo executada. De forma complementar, padrões que permitem a composição recursiva, como o **`Composite`** e o **`Chain of Responsibility`**, possibilitam a combinação do comportamento das implementações. Apesar desse recurso permitir

que seja possível variar, ou mesmo complementar, a funcionalidade de certos pontos de um objeto, ela não permite que sejam adicionadas novas operações.

Esse capítulo irá explorar padrões que podem ser utilizados para a adição de novas funcionalidades e novas operações para classes. Seguindo os princípios da orientação a objetos, nesses casos, a própria operação será considerada uma entidade no software. A partir disso, serão definidas abstrações para que diversas operações possam ser implementadas e incorporadas às classes sem a necessidade de sua modificação.

8.1 Classes que Representam Comandos

"Um comando com sabedoria, é obedecido com alegria."
– Thomas Fuller

Muitas vezes, precisamos definir operações a serem executadas sem que quem aciona essa operação tenha ciência do que está sendo executado. Um exemplo comum disso ocorre em interfaces gráficas, quando algo precisa ser adicionado para ser executado no acionamento de um botão ou de um item de menu. Em outras palavras, é preciso que a execução de uma lógica, que normalmente é representada através de um método, seja representada por uma classe.

Vamos tomar como contexto do exemplo uma aplicação de e-commerce que precisa ser implantada em diversos clientes que possuem diferentes necessidades. O carrinho de compras, por exemplo, que é uma importante classe para o sistema, precisa de diferentes operações dependendo da implantação. Em uma loja que vende produtos de informática, o carrinho deve saber informar o valor do frete e a previsão de entrega, porém essas informações não fazem sentido para uma loja que vende música digital e precisa informar o tamanho dos arquivos para download e o tempo estimado de acordo com a velocidade da conexão do cliente.

Como foi dito na introdução, alguns dos padrões já apresentados nesse livro permitem alterar a implementação de uma operação implementada por uma classe. Porém esse seria um caso diferente, no qual diferentes operações seriam disponibilizadas pela classe, dependendo da situação. Como lidar com isso? Como permitir que as operações disponibilizadas pela classe possam ser incorporadas de forma dinâmica?

Conhecendo o padrão Command

O **Command** é um padrão que consiste na representação de uma operação como uma classe. A abstração que representa o comando, no formato de uma superclasse ou interface, define um método que deve ser utilizado para executar a operação. Os comandos concretos, que implementam ou estendem a abstração, devem definir como atributo a estrutura de dados necessária para a sua execução. Dessa forma, o comando acaba sendo uma classe que pode representar uma operação do sistema independente de todo o contexto. É essa característica que permite diversas aplicações desse padrão!

A Figura 8.1 apresenta a utilização do **Command** no contexto de um componente. Um componente pode possuir como uma *Hook Class* um comando, o qual ele irá executar como resposta a algum evento do sistema. A classe que cria o componente, na figura representada pela classe `Fabrica`, deve injetar a implementação de `Comando` adequada. Nesse caso, o uso de uma interface mais geral permite que uma mesma implementação possa ser usada em vários componentes e que qualquer implementação possa ser associada a um componente específico.

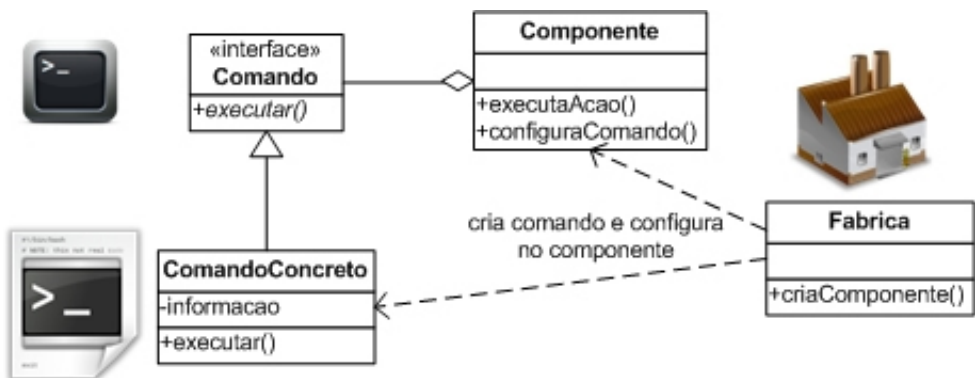



Figura 8.1: Utilização do Command como parte de um componente

Apesar da classe `Componente` representada no diagrama poder ser um componente gráfico ou não, o uso desse padrão é muito comum para a configuração de respostas para ações de usuário para componentes de interface gráfica. Imagine, por exemplo, que um mesmo comando pode ser executado ao se clicar em um botão, ao selecionar um item de menu ou ao acessar um atalho no teclado. Por outro lado, vários botões que são representados pela mesma classe precisam executar diferentes comandos ao serem acionados. Essa liberdade na associação dos comandos aos

componentes é uma das vantagens desse padrão.

Muitas vezes, a classe que possui **Command** não o executa diretamente, mas o passa para outra classe para sua execução. A Figura 8.2 apresenta esse cenário. Nesse caso, a classe `Cliente` pega o comando configurado em algum componente ou cria a instância de `ComandoConcreto` com o comportamento desejado, e

envia a classe `ExecutorComandos` para execução. Essa estrutura faz sentido quando existe a necessidade de um controle central dos comandos que estão sendo executados. Um exemplo dessa situação seria quando é necessário injetar recursos nesso comando, como uma conexão com o banco de dados. Outro cenário seria quando se precisa guardar um histórico dos comandos executados, como para questões de transação ou para possibilitar que o usuário volte atrás em uma ação.

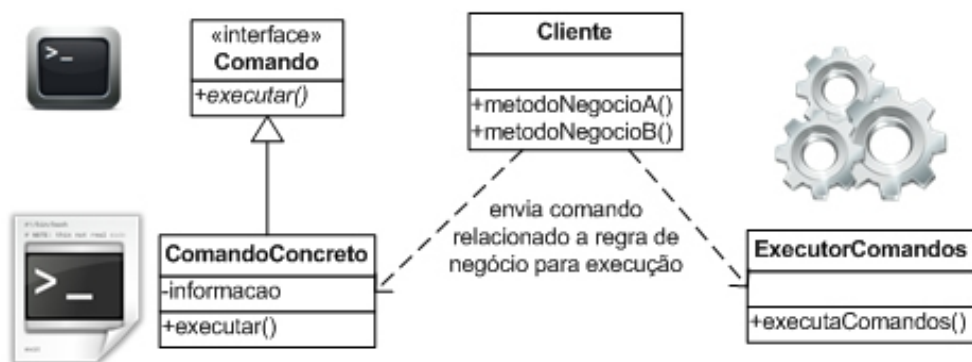



Figura 8.2: Passando o Command como parâmetro para execução

Uma das principais consequências positivas do padrão **Command** está na facilidade de adição de novas operações ao software. Por ele abstrair o conceito de operação, isso permite que novas implementações possam ser facilmente incorporadas. Outro aspecto positivo do **Command** está no fato de ele ser uma representação de uma operação independente do contexto em que ela está inserida. Isso permite que essa operação seja armazenada em um histórico ou enviada pela rede, de forma que suas informações possam ser acessadas e a funcionalidade executada em qualquer momento. Isso é muito útil para a implementação de transações, registro de auditoria e até mesmo a funcionalidade de desfazer ações do usuário.

Por outro lado, o fato de cada operação ser representada por uma classe diferente faz com que o número de classes cresça bastante. Um **Command** também acaba prejudicando o encapsulamento da classe relacionada com sua execução, pois ele precisa

ter acesso às informações dela, que muitas vezes fariam sentido apenas internamente. Uma forma de contornar essa questão é utilizar **Dependency Injection** para injetar as informações nas classes que representam um comando. Uma alternativa para identificar qual parâmetro deve ser injetado poderia ser a utilização da injeção por interfaces, em que a presença de uma interface indicaria a necessidade de uma informação no comando.

Implementando o Carrinho de Compras

Para exemplificar a utilização do padrão **Command**, vamos pegar o exemplo do carrinho de compras utilizado na introdução dessa seção. O objetivo é permitir que o carrinho de compras possa ter diferentes operações de acordo com o contexto em que a aplicação tiver sido implantada. Isso irá permitir não somente que ele possa possuir diferentes operações em diferentes contextos, mas também que novas operações possam ser facilmente incorporadas.

Para implementar o padrão **Command** nesse contexto, as operações do carrinho de compras devem ser representadas pelos comandos. A listagem a seguir apresenta a interface que deve ser implementada pelas classes que representam os comandos. Observe que o método `executar()` não recebe nenhum parâmetro, pois os parâmetros que essa execução tiver deverão ser inseridos na classe e armazenados como atributos. O retorno do tipo `Object` foi utilizado para possibilitar que o comando retornasse qualquer objeto como resultado de sua execução, o qual deve ser tratado por quem invocá-lo.

Listagem 8.1 - Interface que abstrai uma operação do carrinho de compras:

```
public interface ComandoCarrinho{
    public Object executar();
}
```

A classe `CarrinhoCompras`, apresentada na listagem a seguir, é a classe que representa o carrinho de compras propriamente dito. Em sua estrutura de dados, além do usuário e da lista de produtos, também estão os comandos disponíveis para aquela instância. Os comandos são armazenados em um mapa e são identificados por uma String. Dentro desse contexto, o método `executaComando()` seria responsável por executá-lo e passar para ele as informações necessárias para sua execução. Caso o comando não seja encontrado no `Map`, uma exceção do tipo `ComandoNaoEncontradoException` é lançada pelo método.

Listagem 8.2 - Implementação do Carrinho de Compras:

```
public class CarrinhoCompras{

    private Map<String, ComandoCarrinho> comandos;
    private List<Produto> produtos;
    private Usuario usuario;

    //outros métodos de carrinho de compras

    public Object executaComando(String nomeComando)
        throws ComandoNaoEncontradoException{
        ComandoCarrinho c = comandos.get(nomeComando);
        if(c == null)
            throw new ComandoNaoEncontradoException();
        if(c instanceof CienteDosProdutos){
            (CienteDosProdutos(c)).setlistaProdutos(produtos);
        }
        if(c instanceof CienteDoUsuario){
            (CienteDoUsuario(c)).setUsuario(usuario);
        }
        return c.executar();
    }

    public Set<String> getComandosDisponiveis(){
        return comandos.keySet();
    }
}
```

Uma característica interessante dessa solução é a utilização de interfaces para que o carrinho possa injetar as informações nos comandos. As interfaces `CienteDosProdutos` e `CienteDoUsuario` são implementadas pelas classes que desejam respectivamente receber a lista de produtos e as informações do usuário. Isso permite às classes sinalizarem de qual informação precisam para seu processamento e à classe `CarrinhoCompras` saber qual informação ela precisa passar.

Listagem 8.3 - Exemplo de operação que calcula o tamanho para download dos produtos do carrinho:

```
public class TamanhoParaDownload
        implements ComandoCarrinho, CienteDosProdutos{

    private List<Produto> produtos;

    public void setListaProdutos(List<Produto> produtos){
        this.produtos =produtos;
    }

    public Object executar(){
        double tamanho = 0;
        for(Produto p : produtos){
            if(p.isDigital()){
                tamanho += p.getTamanhoDownlaod();
            }
        }
        return double;
    }
}
```

Combinando o Command com Outros Padrões

O padrão **Command** possui estrutura simples, até um pouco parecida com outros padrões baseados em composição, como o **Observer** e o **Strategy**. Devido a essa simplicidade, muitas vezes acaba sendo fácil sua combinação com outros padrões. Por exemplo, quando diversos comandos possuírem passos similares, a utilização de um **Template Method** como base do do método de execução pode evitar a repetição de código. Devido à interface comum compartilhada pelos comandos, a criação de um **Proxy** que encapsule um comando pode ser útil para a adição de funcionalidades em qualquer implementação.

Para que que novos comandos possam ser facilmente adicionados ao sistema, uma combinação comum é a utilização de um **Dynamic Factory** ou de um **Service Locator** para o carregamento das implementações. Muitas vezes, a configuração dos comandos é acompanhada de informações a respeito das situações em que devem ser invocados. A listagem a seguir retoma o exemplo da classe `CarrinhoCompras` e mostra como os comandos poderiam ser carregados dinamicamente no construtor da classe. No caso, o padrão **Service Locator** é utilizado a partir da classe `ServiceLoader`.

Listagem 8.4 - Utilização de um Service Locator para localizar os comandos:

```
public class CarrinhoCompras{

    private Map<String, ComandoCarrinho> comandos;

    public CarrinhoCompras(){
        comandos = new HashMap<>();
        carregarComandos();
    }

    public void carregarComandos(){

    }
}
```

Uma outra situação que pode ocorrer é quando utilizamos o **Command** e há passos similares a serem executados em diversos cenários. Por exemplo, a solicitação de uma confirmação do usuário ou a atualização de uma tabela podem ser passos necessários para diferentes operações. Nesse caso, pode-se considerar a implementação de sequências de comandos, em que o acionamento de um comando resultará na execução de uma cadeia de outros comandos ligados a ele. Somente pela palavra "encadeamento" já é possível pensar no **Chain of Responsibility**!

Para exemplificarmos uma cadeia de comandos, considere a abstração `ComandoCarrinho` apresentada na seção anterior. Para implementar o encadeamento de comandos, iremos transformar a interface em uma classe abstrata, cujo código está apresentado na listagem a seguir. Essa classe abstrata possui o atributo `proximo` que representa o próximo comando da sequência. O método `executar()`, que estava presente na interface, agora se tornou uma método concreto. Esse método invoca primeiramente o método abstrato `executarComando()`, no qual a funcionalidade daquele comando deve ser inserida, e em seguida executa o próximo comando caso esse não seja nulo.

Listagem 8.5 - Interface que abstrai uma operação do carrinho de compras:

```
public abstract class ComandoCarrinho{
    private ComandoCarrinho proximo;

    public ComandoCarrinho(ComandoCarrinho proximo){
        this.proximo = proximo;
    }
```

```
    public void executar(){
        executarComando();
        if(proximo != null)
            proximo.executar();
    }

    protected abstract void executarComando();
    public abstract Object getResultado();
}
```

Outra questão que precisou ser alterada para a implementação da sequência de comandos foi a forma que o resultado do comando é disponibilizado. Na implementação anterior, o retorno era dado pelo próprio método `executar()`. Nesse caso, a não ser que haja uma forma padrão de combinar os resultados da sequência, não é possível retornar o resultado no próprio método. A solução dada nesse caso foi a definição de um método abstrato para o retorno do resultado. Dessa forma, a classe do comando pode escolher como o resultado será obtido e, caso seja aplicável, como ele será combinado com o resultado do comando executado na sequência.

Combine padrões, mas para diferentes problemas

A combinação de padrões é um recurso muito poderoso, que pode gerar uma sinergia que traz benefícios muito maiores do que a utilização isolada de cada padrão. Porém essa combinação deve ser feita com cuidado, pois a utilização de vários padrões em uma mesma solução sem necessidade é uma má prática. Isso porque eles podem tornar a solução complexa sem trazer nenhum benefício. Toda vez que pensar em uma solução com múltiplos padrões, minha sugestão é que você faça um exercício se perguntando qual o problema que cada um está resolvendo. Se caso não conseguir responder essa pergunta para algum deles, considere remover aquele padrão da solução. Mesmo que você esteja errado e o padrão acabe voltando, pelo menos você aprendeu mais sobre o seu problema e pode inclusive considerar outras alternativas.

8.2 Cenários de Aplicação do Command

A seção anterior descreveu o padrão **`Command`** e apresentou o exemplo do carrinho de compras, no qual ele era utilizado para representar novas operações a serem in-

seridas no carrinho. Existem outros cenários importantes onde esse padrão possui uma grande aplicabilidade. É interessante como em cada um esse padrão é implementado por um motivo diferente, porém em todos eles a solução com o **`Command`** é relacionada com o uso de uma classe para representar uma operação. As próximas subseções irão apresentar diferentes ocasiões para o uso, mostrando exemplos de código de como implementá-los.

Execução Remota

Uma das vantagens de se ter uma representação de uma operação do sistema é poder passar essa operação como parâmetro. Esse é o caso mostrado na seção anterior quando havia um executor de comandos. Uma vantagem dessa estrutura é que esse executor pode estar localizado em uma máquina remota, oferecendo serviços para a execução desses comandos. Essa estrutura permite ter uma interface de serviços mais enxuta, provendo para clientes remotos a funcionalidade de execução de comandos.

Para exemplificar como um executor remoto de comandos poderia ser implementado, essa seção mostra o exemplo de um Session Bean que executa comandos remotamente. Primeiro, é preciso definir uma interface remota para essa execução , como mostrado na listagem a seguir. Uma questão notável nesse caso é a interface `Comando` estender `Serializable` para poder ser enviada através da rede. Nesse caso, é importante que as implementações tragam consigo todas as informações necessárias para a execução.

Listagem 8.6 - Interface remota pra execução de comandos:

```
@Remote
public interface ExecutorComandos{
    public Object executarComando(Comando c);
}
```

Em seguida, precisamos implementar o Session Bean que implementa a interface remota definida. Isso é apresentado na listagem a seguir. O método `executarComando()` executa o comando e em seguida retorna o resultado da execução que é recuperado de uma propriedade desse objeto. Esse método também pode injetar no comando recursos que estão presentes apenas no servidor, como o `EntityManager` mostrado no exemplo, que serve para acessar o banco de dados. Para isso, basta o comando implementar a interface `PrecisaEntityManager`.

Listagem 8.7 - Session Bean para execução remota de comandos:

```
@Stateless
public class ExecutorRemoto implements ExecutorComandos{
    @PersistenceContext
    private EntityManager em;

    public Object executarComando(Comando c){
        if(c instanceof PrecisaEntityManager){
            ((PrecisaEntityManager)c).setEntityManager(em);
        }
        c.executar()
        return c.getResultado();
    }
}
```

A implementação de um cliente para esse serviço acaba sendo muito simples. A listagem a seguir mostra um método utilitário que poderia ser usado para a execução de comandos em um servidor remoto. As propriedades passadas para o método servem, a grosso modo, para a localização do servidor e o parâmetro de nome `jndiName` para a localização do serviço de execução de comandos. Como essas informações dependem do servidor e da sua versão, decidiu-se por passá-las como parâmetro. A partir da chamada desse método o comando será enviado ao servidor e executado, e o resultado de sua execução enviado de volta e retornado.

Listagem 8.8 - Cliente que acessa o Session Bean para execução de um comando no servidor:

```
public static Object executaComandoRemoto(Comando c,
        String jndiName, Properties props) throws NamingException{

    InitialContext ic = new InitialContext(props);
    ExecutorComandos ec = ic.lookup(jndiName);
    return ec.executarComando(c);
}
```

A principal vantagem dessa solução é que fica muito simples adicionar novas funcionalidades para serem executadas no servidor. Normalmente, cada nova operação no servidor precisa da definição de um método na interface remota e na implementação para que seja consumido por seus clientes. Na solução apresentada,

apenas o método de execução de comandos precisa ser exposto remotamente, bastando a criação de um novo comando para a introdução de uma operação.

Uma consequência negativa dessa interface mais geral é que isso facilita a introdução de comandos maliciosos, abrindo uma brecha para execução de funcionalidades diversas no servidor. Apesar da classe do comando precisar estar presente no *classpath* do servidor para poder ser executada, é aconselhável implementar no servidor funcionalidades de segurança visando autorizar o acesso apenas a comandos conhecidos.

Transações e Logging

O fato de as operações serem representadas por objetos e poderem ser armazenadas permite que seja feito um histórico dos comandos executados pelo sistema. Ele pode ser persistido, servindo como um log do que foi executado pela aplicação. Isso também possibilita que essas operações sejam refeitas no caso de uma queda do sistema, o que faz com que os comandos sirvam para representar transações duráveis em sistemas de informação.

Para ilustrar o uso do **`Command`** para implementação de transações, será utilizado como exemplo o framework Prevayler (http://prevayler.org) . Esse framework é uma implementação de código aberto que faz a persistência de objetos em Java. Com a implementação do padrão **`Prevalent System`** [26], ele mantém os objetos persistentes em memória e armazena as transações para uma possível recuperação do sistema. De tempos em tempos, é tirado um snapshot do objeto persistido, que é gravado serializado em disco. Nos intervalos entre entre essas gravações, as transações são armazenadas para guardar as alterações realizadas.

Considere, por exemplo, que o objeto que está sendo persistido pelo Prevayler seja uma lista e que uma das operações que podem ser feitas nessa lista seja a inserção de um novo elemento. A listagem a seguir apresenta como essa transação seria implementada. Observe que ela possui como parte de sua estrutura de dados o item que será inserido, além do método de execução definido na interface `Transaction`, no caso `executeOn()`.

Listagem 8.9 - Exemplo de uma transação do Prevayler para adicionar um item em uma lista:

```
public class InserirItem implements Transaction<List<Serializable>>{

    private Serializable item;
```

```
    public InserirItem(Serializable item){
        this.item = item;
    }
    public void executeOn(List<Serializable> lista, Date date){
        lista.add(item);
    }
}
```

A utilização do **Command** para a definição de transações, nesse caso, permite que ela seja armazenada e recuperada se necessário. Caso a lista fosse modificada diretamente, seria preciso armazenar essa lista a cada alteração para que elas não fossem perdidas. Como é possível persistir a alteração, é possível armazenar uma versão da lista e todos os comandos executados nela a partir daquele ponto. Dessa forma, caso seja necessário recuperar sua última versão, basta recuperar a versão serializada com todas as transações a partir daquele ponto e executar as transações novamente.

Fazer e Desfazer

Uma outra aplicação do **Command** é para dar suporte na aplicação à funcionalidade de desfazer ações realizadas pelo usuário. Nesse caso, as operações realizadas por ele devem ser armazenadas para poderem ser acessadas e desfeitas. Para que isso seja possível, além de um método para executar o comando, também é necessário um que reverta os seus efeitos. A listagem a seguir mostra o exemplo de uma interface para a definição de um comando com os métodos `fazer()` e `desfazer()`.

Listagem 8.10 - Interface para comando que pode ser desfeito:

```
public interface Comando{
    public void fazer();
    public void desfazer();
}
```

Este cenário é um exemplo no qual uma classe para a execução de comandos se faz necessária. Seu papel é prover um local centralizado para execução e armazenamento dos comandos, para viabilizar que eles possam ser recuperados e revertidos se necessário. A listagem a seguir apresenta um executor de comandos que permite que os comandos executados possam ser desfeitos e que os comandos desfeitos

possam ser refeitos.

Listagem 8.11 - Executor de comandos que possibilita que comandos sejam desfeitos e refeitos:

```
public class ExecutorComandos {
    private Queue<Comando> feitas;
    private Queue<Comando> desfeitas;

    public ExecutorComandos() {
        feitas = new LinkedList<>();
        desfeitas = new LinkedList<>();
    }
    public void executarComando(Comando c) {
        c.fazer();
        feitas.offer(c);
        desfeitas.clear();
    }
    public void desfazer() {
        if(!feitas.isEmpty()) {
            Comando c = feitas.pool();
            c.desfazer();
            desfeitas.offer(c);
        }
    }
    public void refazer() {
        if(!desfeitas.isEmpty()) {
            Comando c = desfeitas.pool();
            c.fazer();
            feitas.offer(c);
        }
    }
}
```

Essa classe possui duas pilhas que são chamadas de `feitas` e `desfeitas`. Quando um comando é executado, ele é empilhado na pilha `feitas`. Quando o método `desfazer()` é invocado, o comando de cima da pilha `feitas` é desempilhado, invoca-se nele o método `desfazer()` e ele é colocado na pilha `desfeitas`. Nesse momento, o usuário ainda tem a oportunidade de voltar atrás com a chamada do método `refazer()`, o qual desempilha um comando da pilha `desfeitas`,

executa o comando e o adiciona na pilha `feitas`. Porém, como só faz sentido refazer um comando logo que foi desfeito, sempre que um novo comando é executado, a pilha `desfeitas` é esvaziada.

Em um executor de comandos mais sofisticados, poderia haver comandos que não podem ser defeitos ou comandos inertes, ou seja, que não deveriam ser incluídos na pilha de comandos realizados. Com pequenas modificações, a implementação apresentada poderia ser adaptada para esse tipo de requisito. Nesse caso de haver diversos tipos de comando, diferentes interfaces que expressam as possíveis operações de cada um deles poderiam ser utilizadas para diferenciação. Outra alternativa seria a utilização de uma interface única e de anotações para marcar os comandos que precisam ser tratados de forma distinta.

8.3 Double Dispatch - Me chama, que eu te chamo!

"Não ligue para nós, nós ligamos para você!"
– Princípio de Hollywood

Uma situação que muitas vezes ocorre no cotidiano é uma pergunta ser respondida com outra pergunta. Imagine que uma garota questione o seu namorado: "*Eu vi você dentro da boate na sexta a noite! O que você estava fazendo lá?*" e ele responda: "*Se você me viu você também estava lá! O que você está fazendo lá?*". Brincadeiras à parte, uma técnica importante que pode ser utilizada em um projeto orientado a objetos é fazer com que uma chamada de método seja respondida com a chamada de outro método no objeto passado como parâmetro.

Para ilustrar o problema, vamos retomar o exemplo do carrinho de compras, porém considerando que diversos tipos produtos possam ser adicionados no carrinho em uma mesma loja. Dependendo do tipo de produto, diferentes propriedades precisam ser modificadas. Uma música, por exemplo, adicionaria seu preço no valor total e seu tamanho no total para download. Já um produto físico, além do preço, adicionaria seu peso e volume, que seriam utilizados posteriormente para o cálculo do frete. Além disso outros tipos de produto podem surgir, como a venda de passagens aéreas ou entradas para eventos. Vamos ver nessa seção como seria a solução sem a utilização de padrões e depois com o uso do **`Double Dispatch`** [28].

Resolvendo o Problema do Carrinho sem Padrões

O primeiro instinto para a implementação da adição de diferentes tipos de produtos seria criar condicionais para selecionar entre os vários tipos de produto. Nesse

caso, o tipo do produto, através da sua classe, seria utilizado para diferenciá-los. A listagem a seguir mostra o exemplo de como ficaria o código dessa solução. A partir dela, a cada novo tipo de produto, um novo condicional precisaria ser adicionado nesse método.

Listagem 8.12 - Usando condicionais para identificar o tipo do produto:

```
public class CarrinhoCompras {
    //outros métodos e atributos

    public void adicionarProduto(Produto p) {
        produtos.add(p);
        if(Produto instanceof ProdutoDigital) {
            ProdutoDigital pd = (ProdutoDigital) p;
            adicionaPropriedade("PRECO", pd.getPreco());
            adicionaPropriedade("DOWNLOAD", pd.getTamanho());
        } else if(Produto instanceof ProdutoFisico) {
            ProdutoFisico pf = (ProdutoFisico) p;
            adicionaPropriedade("PRECO", pf.getPreco());
            adicionaPropriedade("VOLUME", pf.getVolume());
            adicionaPropriedade("PESO", pf.getPeso());
        } else {
            //mais condicionais para cada tipo de produto
        }
    }
}
```

Uma forma de eliminar os condicionais seria a criação de um método para a adição de cada tipo de produto, como mostrado na próxima listagem. Como em Java pode-se fazer a sobrecarga de métodos, é possível que todos possuam o mesmo nome. Dessa forma, o método seria selecionado de acordo com o tipo de produto passado para ele. Apesar dessa solução eliminar os condicionais, ainda seria necessária a adição de novos métodos na classe `CarrinhoCompras` para a adição de novos tipos produtos.

Listagem 8.13 - Usando um método diferente para cada tipo de produto:

```
public class CarrinhoCompras {
    //outros métodos e atributos
```

```
    public void adicionarProduto(ProdutoDigital pd) {
        produtos.add(pd);
        adicionaPropriedade("PRECO", pd.getPreco());
        adicionaPropriedade("DOWNLOAD", pd.getTamanho());
    }
    public void adicionaProduto(ProdutoFisico pf) {
        produtos.add(pf);
        adicionaPropriedade("PRECO", pf.getPreco());
        adicionaPropriedade("VOLUME", pf.getVolume());
        adicionaPropriedade("PESO", pf.getPeso());
    }
    //outros métodos para cada tipo de produto
}
```

O problema aqui é que a adição de um novo tipo de produto na verdade é uma nova operação para a classe `CarrinhoCompras`, pois cada um possui uma lógica diferente. Ambas as implementações apresentadas exigem que mais código seja adicionado na classe para a adição ao suporte de um novo tipo de produto. Como permitir isso sem alterar a classe?

Aplicando o Double Dispatch no Carrinho de Compras

Como talvez você desconfie pela introdução, a melhor forma de responder essa pergunta é devolver a pergunta! Nesse caso, a pergunta que está sendo feita para o carrinho é o que ele precisa fazer quando um determinado tipo de produto for adicionado. Então, uma forma de devolver essa pergunta seria ele questionar o produto qual propriedade ele deve adicionar no carrinho. A listagem a seguir mostra como ficaria a classe `CarrinhoCompra` com a implementação dessa solução.

Listagem 8.14 - Utilizando o Double Dispatch no carrinho de compras:

```
public class CarrinhoCompras {
    //outros métodos e atributos

    public void adicionarProduto(Produto p) {
        produtos.add(p);
        p.adicionaPropriedades(this);
    }
}
```

Observe que o método `adicionarProduto()` recebe uma instância da classe `Produto` como parâmetro e devolve para essa instância a invocação do método `adicionaPropriedades()`, passando-se como parâmetro. Abaixo segue o exemplo de como seria a implementação da classe `ProdutoDigital`, que agora tem a responsabilidade de adicionar suas responsabilidades no carrinho. Observe que o método recebe o próprio carrinho de compras como parâmetro e adiciona as propriedades adequadas. Caso alguma outra modificação no `CarrinhoCompras` fosse necessária, ela também poderia ser feita nesse método.

Listagem 8.15 - Exemplo de tipo de produto com o uso do Double Dispatch:

```
public class ProdutoDigital extends Produto {
    //atributos de um produto digital

    @Override
    public void adicionaPropriedades(CarrinhoCompras c) {
        c.adicionaPropriedade("PRECO", getPreco());
        c.adicionaPropriedade("DOWNLOAD", getTamanho());
    }
}
```

Essa é justamente a ideia da estrutura do padrão **Double Dispatch**: a classe devolve a chamada de método para o parâmetro que recebe. Assim, dependendo do tipo passado como parâmetro, o método invocado terá uma implementação diferente. Observe que antes da implementação do padrão, era necessária a modificação da classe `CarrinhoCompras`, por meio da adição de um método ou da criação de um condicional, para dar suporte a um novo tipo de produto. Sendo assim, a cada novo tipo de produto, é como se tivéssemos adicionando um novo tipo de operação na classe.

Estrutura do Double Dispatch

Se formos traduzir de forma literal do inglês, o nome do padrão **Double Dispatch** significa "despacho duplo". Nesse contexto, a palavra "despacho" significa a delegação de parte da funcionalidade da classe para outra. A adição da palavra "duplo", significa que essa delegação ocorre duas vezes. Sendo assim, o nome do padrão acaba ressaltando sua principal característica, que é a da classe que recebe a chamada do método devolvê-la para um objeto recebido como parâmetro, o qual irá executar uma operação na classe que chamou o método.

A Figura 8.3 representa a estrutura do padrão. Uma característica que precisa ser ressaltada é a dependência cíclica entre as classes `Despachante` e `Elemento`, em que cada classe possui um método que recebe a outra como parâmetro. Apesar disso, é importante notar que mesmo que `Despachante` dependa de `Elemento`, ela não depende de nenhuma de suas implementações. Dessa forma, qualquer uma delas pode ser passada como parâmetro e novas implementações podem ser facilmente incorporadas.

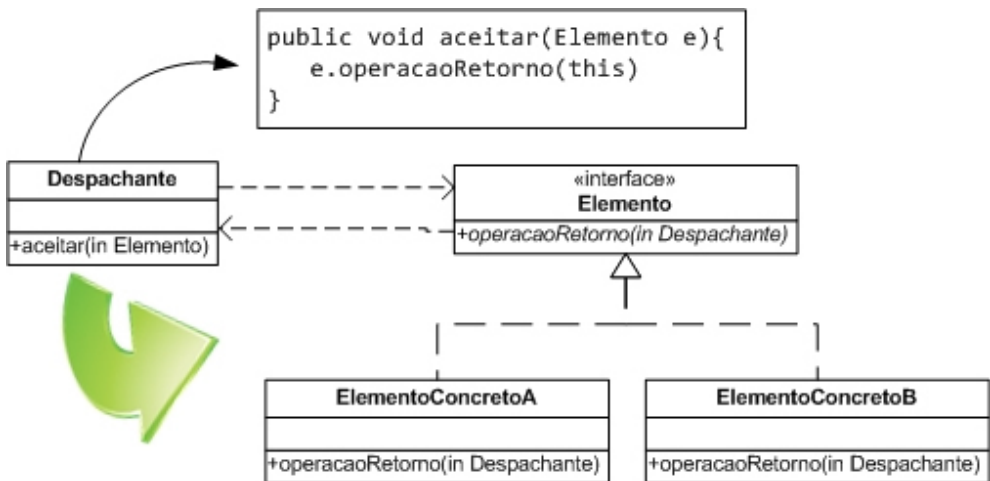



Figura 8.3: Estrutura básica do Double Dispatch

Uma visão dinâmica desse processo do **`Double Dispatch`** está na Figura 8.4. No primeiro caso, quando o elemento é passado como parâmetro, ele recebe uma chamada da classe `Despachante` e executa uma operação nessa classe. No segundo caso, quando um elemento diferente é passado como parâmetro, operações diferentes são executadas na classe `Despachante`. Isso faz com que seja possível determinar a operação a ser executada a partir do parâmetro passado.

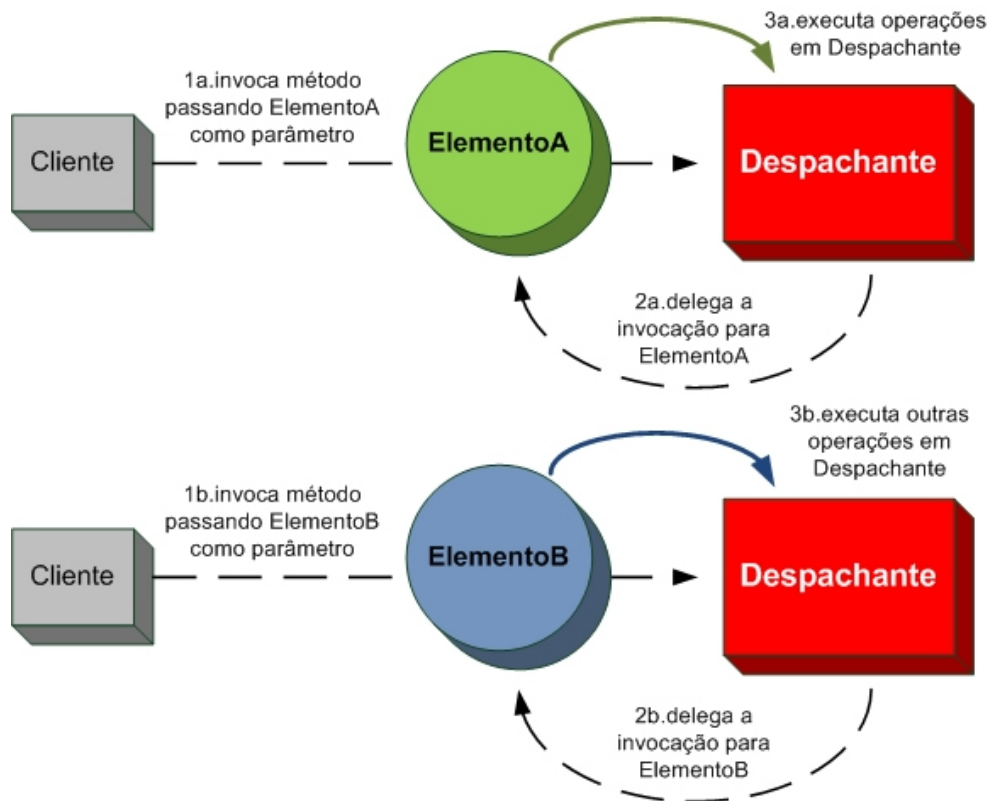



Figura 8.4: Visão dinâmica do funcionamento do Double Dispatch

Cada classe diferente passada como parâmetro é como se fosse um método diferente sendo executado na classe `Despachante`. Sendo assim, o **`Double Dispatch`** é um padrão que pode ser utilizado quando se deseja acrescentar novas operações em uma classe de forma dinâmica. Para isso, basta implementar a interface e criar a nova operação no método que é chamado de volta pela classe.

8.4 Padrão Visitor

"O Visitor é um padrão de que você dificilmente precisa, mas, quando precisa, nenhuma outra solução resolve o problema."
– Joseph Yoder

O padrão **`Visitor`** na minha opinião é um dos mais difíceis de ser compreendidos e aplicados do GoF [10]. Em cursos sobre modelagem orientada a objetos que ministro, algumas vezes fiz um workshop de padrões, em que cada aluno apresentava um padrão. Com muita frequência, quem apresentava o **`Visitor`** acabava se confundindo e entendendo errado sua idea. Muitos dos exemplos que encontravam na Internet eram tão artificiais que acabavam ilustrando somente a estrutura, mas não a motivação para o uso do padrão.

O **`Visitor`** utiliza em sua estrutura o **`Double Dispatch`**, visto na seção anterior. A compreensão desse padrão certamente já é um passo importante para a compreensão do **`Visitor`**. A próxima seção irá apresentar o exemplo de um problema que dificilmente teria uma solução eficiente sem a utilização desse padrão. Em seguida será mostrado como ele funciona e como poderia ser utilizado para resolver o problema.

Diferentes Relações entre Padrões

Até o momento já vimos diversas formas de relacionamento entre padrões. O exemplo mais simples seria quando dois padrões podem ser utilizados em conjunto para a solução de um problema. Outro exemplo, seria quando dois ou mais fornecem soluções alternativas para o mesmo problema. No caso do **`Visitor`** e do **`Double Dispatch`** temos um novo tipo de relação: quando a solução de um padrão é utilizada como parte da solução de outro.

Por mais que isso possa parecer estranho, esse tipo de relação no mundo dos padrões é natural. Existem padrões mais fundamentais e mais gerais que são utilizados por outros que resolvem problemas mais específicos. No livro "Implementation Patterns" [2], por exemplo, são apresentadas diversas práticas mais granulares que acabam sendo utilizadas dentro dos padrões apresentados nesse livro. É importante ressaltar que ainda existem várias outras formas de relacionamento entre eles [18], além das apresentadas nesse livro.

Vários Relatórios em Diversos Formatos

O exemplo dessa seção irá utilizar como contexto um software que precisa gerar diversos tipos de relatório em diversos formatos. Em relação aos tipos de relatório,

cada um possui informações e uma lógica de montagem diferente. Podemos ter um que sumariza as compras de um cliente e outro que mostra as vendas mensais de um determinado produto. Outras vezes, a mesma informação pode ser exibida de formas diferentes, como com gráficos ou com tabelas. Sendo assim, apesar dos elementos de um relatório serem os mesmos, como texto, tabelas, títulos e gráficos, suas informações e sua estrutura são diferentes.

Em relação ao formato em que o relatório será gerado, também existem várias possibilidades. Exemplos de formatos que podem ser necessários são em HTML para visualização em browsers, em PDF para permitir uma melhor leitura e impressão, planilha para permitir uma manipulação posterior dos dados e até mesmo em XML para que o relatório possa ser interpretado e exibido por outras aplicações. Para a geração em cada um dos formatos, uma API diferente precisa ser utilizada pela aplicação, como o iText para PDF e o Jakarta POI para planilhas. Sendo assim, mesmo que os elementos dos relatórios sejam os mesmos, a geração de cada um deles em cada plataforma é completamente diferente.

A Figura 8.5 ilustra a relação de muitos para muitos entre relatórios e os formatos. Como cada um possui informações diferentes, fica difícil reutilizar o código de geração de um para outro. Adicionalmente, como cada API cria os elementos de forma diferente, também não é possível reutilizar o código de geração para diferentes formatos dentro do mesmo relatório.

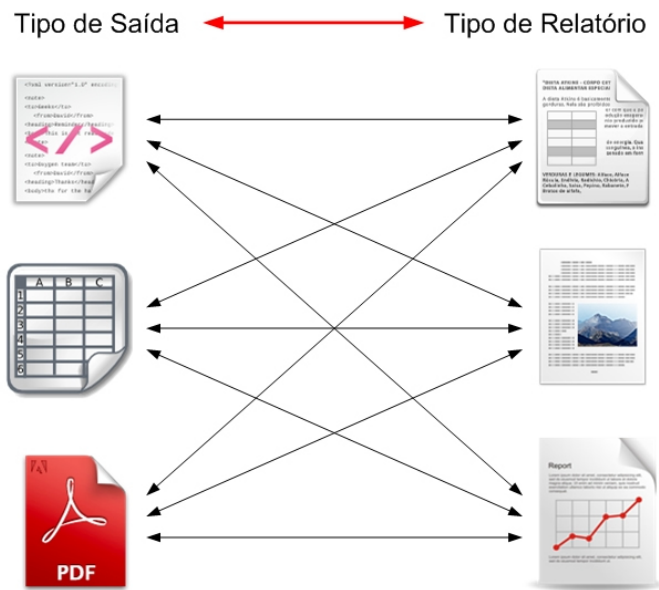



Figura 8.5: Relacionamento entre os tipos de relatório e os tipos de saída

Como essa reutilização de código não é direta, a solução comum nesse caso seria cada um dos relatórios possuir um método para a geração em cada formato, como mostrado na Figura 8.6. Dessa forma, a cada novo relatório, precisaria ser criado um método para a sua geração em cada um dos formatos. Se um novo formato fosse introduzido, isso também demandaria a criação de um novo método em cada classe de relatório.

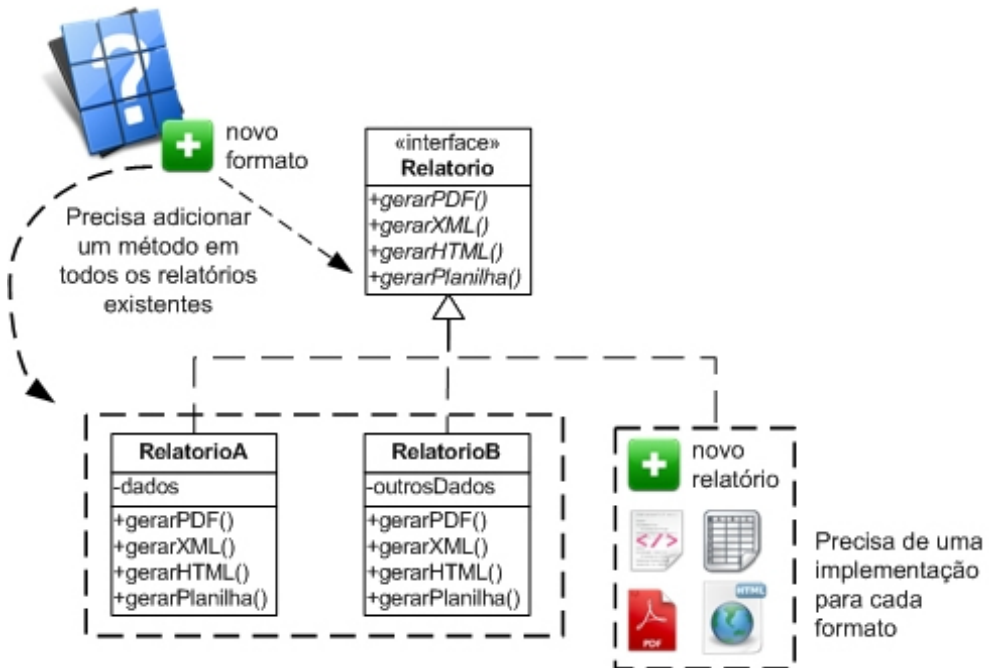



Figura 8.6: Estrutura da solução do problema do relatório sem o uso de padrões

O problema não pára na introdução de novos elementos, mas também aparece na manutenção dos formatos e relatórios já existentes. Imagine, por exemplo, que como um novo requisito do usuário seja necessário a adição do logo da empresa no início dos relatórios em PDF. Isso vai demandar uma atualização em todas as classes de relatório. Por outro lado, se fosse necessária a introdução de uma nova informação no relatório, todos os métodos precisariam ser atualizados.

Introduzindo o Visitor

Para começar a explicar o `Visitor`, vou utilizar uma metáfora que ilustra bem como o padrão funciona. Imagine que existam duas pintoras, Maria Eduarda e Ana Beatriz, sendo que a primeira possui um estilo de pintura mais clássico e a segunda um estilo mais cubista. Considere também que elas vão visitar dois locais diferentes, sendo um deles uma favela e o outro uma casa no campo. Quando forem na favela, será pedido a elas *"pinte essa favela"* e quando forem a casa de campo será pedido *"pinte essa casa de campo"*.

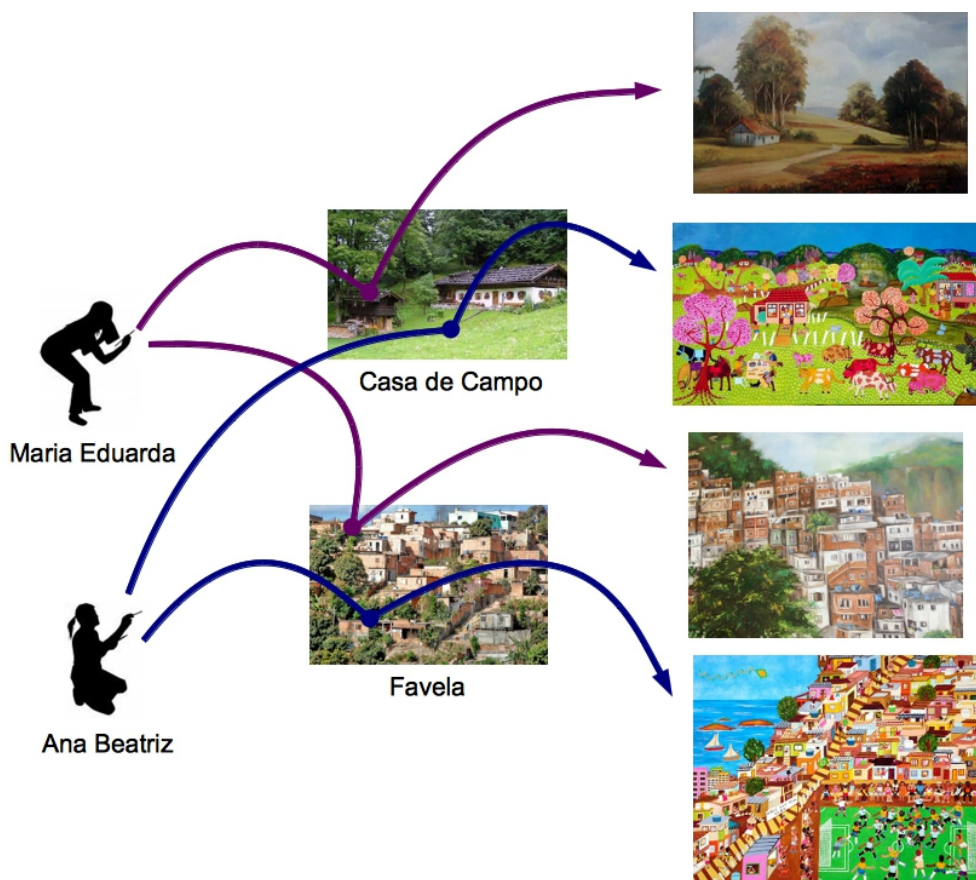
A Figura 8.7 apresenta como foi o resultado das visitas das pintoras aos locais. Apesar do pedido feito a elas ao visitarem um local ser o mesmo, o resultado do pedido, no caso o quadro, será totalmente diferente. Como cada pintora possui um estilo diferente, a execução do mesmo pedido será distinta. Sendo assim, o resultado final depende tanto do local que está sendo visitado, quanto da pintora que o está visitando.

Quadros:
"Futebol na Favela" e "Parada para Merenda" de Helena Coelho e
"Favela" e "Casa de Campo" de Sury Peralles.

Figura 8.7: Visita de pintoras diferentes em locais diferentes

O **Visitor** funciona de forma análoga ao exemplo das pintoras. Quando esse

padrão é utilizado, existe uma hierarquia de classes que representa os visitantes e uma outra que representa elementos que aceitam visitantes. Quando um elemento recebe um visitante, solicita uma tarefa a ele, passando-se como parâmetro ou passando alguma informação que possui internamente. Assim, cada classe que implementa a abstração de visitante irá realizar essa tarefa de um modo diferente e cada implementação de um elemento irá solicitar algo diferente aos visitantes.

A estrutura do **Visitor** está apresentada na Figura 8.8. A interface `Visitante` define quais métodos devem ser implementados pelos visitantes. De forma complementar, a interface `Elemento` define um método para aceitar uma visita. Desse modo, durante ela, cada implementação de `Elemento` irá chamar um método de `Visitante` diferente. De maneira análoga ao exemplo das pintoras, o resultado depende tanto do elemento visitado quanto do visitante. É como se cada implementação de `Visitante` fosse uma nova operação sendo adicionada para o conjunto de classes que implementam `Elemento`.

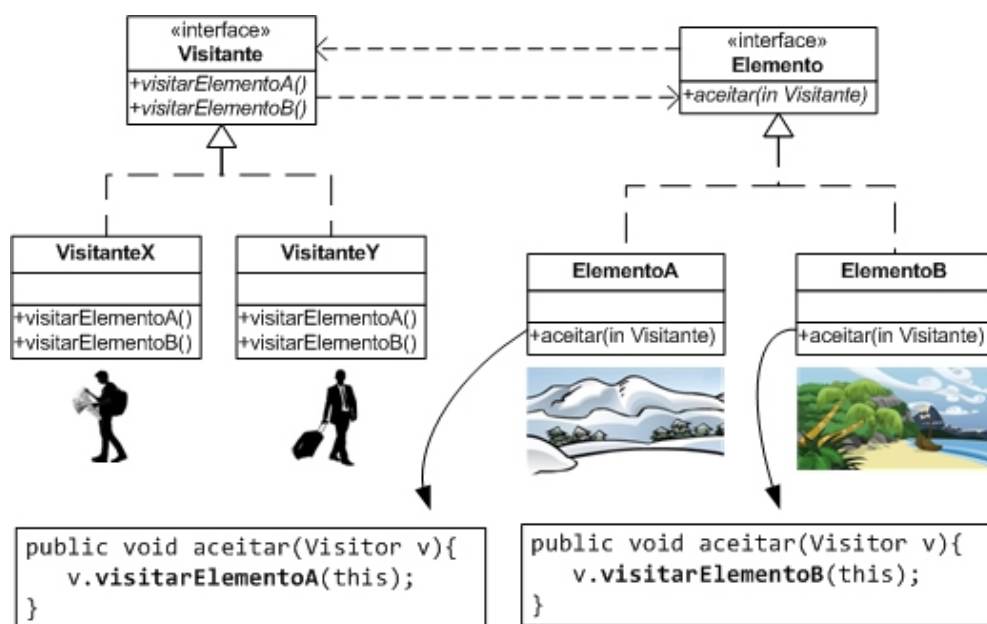



Figura 8.8: Estrutura do padrão Visitor

A representação clássica do **Visitor** é como a apresentada na Figura 8.8, na qual a interface `Visitante` possui um método para cada implementação da inter-

face `Elemento`. Assim fica bem claro qual método uma determinada implementação irá chamar; porém, para cada nova implementação de `Elemento`, um novo método precisaria ser adicionado em cada um dos visitantes. Dessa forma, ficaria fácil adicionar um novo visitante mas não um novo elemento.

Apesar do **`Visitor`** ser frequentemente retratado com essa estrutura, nem sempre os métodos definidos na interface `Visitante` precisam refletir as implementações de `Elemento`. Esses métodos podem definir as operações que podem ser feitas por um visitante e o que vai diferenciar cada elemento será a sequência que irão chamar e os parâmetros que irão passar para cada método. Na próxima seção, o exemplo da geração de relatórios em diferentes formatos irá utilizar essa abordagem.

O diagrama de sequência da Figura 8.9 mostra como um cliente invocaria uma operação a partir do **`Visitor`**. Inicialmente o cliente criaria a instância da implementação desejada da interface `Visitante` e a passaria como parâmetro para o método `aceitar()` de um `Elemento`. Dentro desse método, o método `visitar()` do `Visitante` seria invocado passando a própria classe ou informações como parâmetro. Esse método por sua vez, invocaria métodos na classe passada como parâmetro. Por esse diagrama de sequência fica bem claro a utilização do **`Double Dispatch`** como parte da solução do **`Visitor`**.

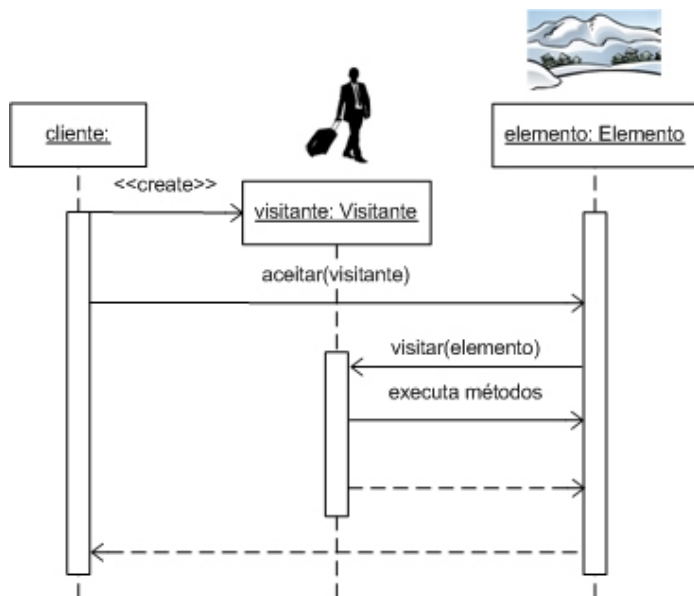



Figura 8.9: Sequência de invocação em um Visitor

A principal diferença entre o **Visitor** e o **Double Dispatch** está na hierarquia dos elementos. No **Double Dispatch**, operações podem ser adicionadas através da passagem de uma implementação diferente como parâmetro. A flexibilidade está no método do tipo do parâmetro para o qual é delegada a execução. Já no **Visitor**, existe também diversas possibilidades de implementação para a classe que aceita o parâmetro. Dessa forma, cada implementação pode invocar métodos diferentes no objeto recebido como argumento.

Fazendo os Formatos de Saída Visitarem os Relatórios

Depois de ver como funciona o **Visitor**, esta seção apresenta como ele pode ser utilizado no problema da geração de relatórios em diversos formatos. Dentro do contexto do padrão, o relatório representará o elemento, e o formato do relatório será o visitante. Quando o método de geração do relatório for chamado passando o visitante como parâmetro, os seus métodos serão chamados com os dados do relatório, de forma a criar a sua estrutura.

A listagem a seguir apresenta a interface que define os métodos que um visitante deve implementar. Observe que cada método representa uma seção do relatório, com a exceção de `getResultado()` que é utilizado no final para retornar o resultado da geração. Dessa forma, cada implementação deve criar uma lógica que crie a seção do relatório adequada. Por exemplo, enquanto um relatório irá gerar um título dentro de um documento PDF, o outro irá criar as tags para a representação do título em uma página HTML.

Listagem 8.16 - Interface implementada pelas classes que representam um formato de arquivo:

```
public interface FormatoVisitante{
    public void visitarTitulo(String t);
    public void visitarSubtitulo(String t);
    public void visitarParagrafo(String p);
    public void visitarTabela();
    public void visitarTabelaCabecalho(String... ct);
    public void visitarTabelaLinha(Object... o);
    public void visitarTabelaFim();
    public void visitarImagem(String path);
    public Object getResultado();
}
```

Conforme a listagem a seguir, a interface `Relatorio`, ao invés de possuir diversos métodos como na solução representada na Figura 8.6, agora possui apenas um método chamado `gerarRelatorio()`. Esse método recebe uma classe do tipo `FormatoVisitante` como parâmetro, que é quem irá determinar o formato de geração do relatório. Enquanto a solução anterior possuía um método para cada formato, nesse caso, cada tipo de `FormatoVisitante` passado como parâmetro pode ser utilizado como a definição de uma nova operação nessa classe.

Listagem 8.17 - Interface implementada pelos relatórios:

```
public interface Relatorio{
    public Object gerarRelatorio(FormatoVisitante fv);
}
```

Para ilustrar a geração de um relatório, a listagem a seguir mostra como seria criado um relatório que listasse as compras de um cliente. Observe que a criação das seções do relatório, como título, parágrafo e tabela, se dá a partir da chamada dos métodos do objeto do tipo `FormatoVisitante` passado como parâmetro. Dessa forma, cada relatório diferente iria chamar esses métodos na ordem necessária, passando seus dados como parâmetro. Nesse caso, não existe um método específico para cada tipo de relatório.

Listagem 8.18 - Implementação da geração de um relatório:

```
public class ComprasCliente implements Relatorio {
    private Cliente c;
    private List<Item> items;

    //outros métodos

    public Object gerarRelatorio(FormatoVisitante fv) {
        fv.visitarTitulo("Compras de "+c.getNome());
        fv.visitarParagrafo("CPF "+ c.getCPF());
        fv.visitarParagrafo("Cliente desde "+ c.getDataCadastro());
        fv.visitarTabela();
        fv.visitarTabelaCabecalho("Produto", "Data", "Valor");
        for(Item i : items) {
            fv.visitarTabelaLinha(i.getProduto(),
                i.getDataCompra(), i.getValor());
        }
```

```
        fv.visitarTabelaFim();
        return fv.getResultado();
    }
}
```

Ainda exemplificando a implementação das interfaces, a listagem a seguir mostra como seria uma implementação de `FormatoVisitante` para a geração de relatórios em HTML. A variável `sb`, do tipo `StringBuilder`, é utilizada para guardar o relatório entre as chamadas de método. Dessa forma, a cada chamada de método, uma nova parte do relatório é adicionada ao `StringBuilder`. De forma similar, outras implementações armazenariam o relatório que estaria sendo gerado em uma variável, e iriam agregando as seções a ele a medida que os métodos fossem chamados.

Listagem 8.19 - Implementação do formato HTML para relatórios:

```
public class VisitanteHTML implements FormatoVisitante {
    private StringBuilder sb = new StringBuilder();

    public void visitarTitulo(String t){
        sb.append("<h1>"+t+"</h1>");
    }
    public void visitarSubtitulo(String t){
        sb.append("<h2>"+t+"</h2>");
    }
    public void visitarParagrafo(String p){
        sb.append("<p>"+t+"</p>");
    }
    public void visitarTabela(){
        sb.append("<table>");
    }
    public void visitarTabelaCabecalho(String... ct){
        sb.append("<tr>");
        for(String s : ct)
            sb.append("<th>"+s+"</th>");
        sb.append("</tr>");
    }
    public void visitarTabelaLinha(Object... obs){
        sb.append("<tr>");
        for(Object o : obs)
            sb.append("<td>"+o.toString()+"</td>");
```

```
        sb.append("</tr>");
    }
    public void visitarTabelaFim(){
        sb.append("</table>");
    }
    public void visitarImagem(String path){
        sb.append("<img src='"+path+"'>");
    }
    public Object getResultado(){
        return sb.toString();
    }
}
```

Para finalizar o exemplo, a listagem a seguir apresenta o código que seria usado para a geração de um relatório. Inicialmente a classe do relatório e do formato precisariam ser criadas. De alguma forma as informações seriam recuperadas, estando isso fora do escopo do exemplo. E então para a sua geração, bastaria a chamada do método `gerarRelatorio()` passando a implementação do visitante como parâmetro.

Listagem 8.20 - Código cliente para a geração de relatórios:

```
Relatorio r = new ComprasCliente();
//carrega dados no relatório

FormatoVisitante fv = new VisitanteHTML();
String resultado = (String) r.gerarRelatorio(fv);
```

Essa solução permite que novos relatórios sejam facilmente adicionados, não importando o formato de geração, e que novos formatos sejam adicionados, já funcionando para todos os relatórios. A grande dificuldade de manutenção do **Visitor** ocorre quando é necessária a inserção de um novo método na interface da classe visitante. Nesse caso, isso aconteceria se fosse necessária a geração de um novo tipo de seção para os relatórios. A dificuldade ocorreria devido ao fato de ser necessária a criação de um novo método que precisaria ser adicionado em todas as implementações.

Elementos Compostos no Visitor

Um padrão que se encaixa muito bem com o **`Visitor`** é o **`Composite`**. Nesse caso, o elemento que recebe o visitante seria composto por outros elementos que também podem receber o visitante. Nesse caso, o elemento composto poderia realizar chamadas próprias no visitante e delegar parte da execução para os elementos que o compõe, passando o visitante para que eles realizem as próprias chamadas.

Dentro do exemplo dos relatórios, um deles poderia ser composto por outros. Dessa forma, uma parte que fosse recorrente, poderia ser modelada como um relatório à parte e utilizada para compor outros relatórios. Sendo assim, o método que aceita um visitante, além de ser implementado pelo elemento principal, também pode ser implementado pelas classes que o compõem.

8.5 Considerações Finais do Capítulo

Esse capítulo abordou um problema importante para o desenvolvimento de aplicações, que é a adição de operações de forma dinâmica. Em alguns casos, a adição de operações em objetos faz parte do próprio negócio e a estrutura do sistema precisa permitir a sua evolução nesse sentido, sem que seja necessária a modificação frequente das classes envolvidas.

O primeiro padrão apresentado nesse capítulo foi o **`Command`**. Ele consiste no encapsulamento de uma operação como uma classe, em vez de sua representação através de um método. A partir dessa estrutura, é possível compor as classes dinamicamente com comandos, tornando possível a adição de novas operações. Outra possibilidade trazida por esse padrão é de armazenar os comandos já executados, dessa forma pode-se utilizá-los para a implementação de transações e auditoria, e até mesmo guardá-los para que possam ser desfeitos.

Outros padrões para a adição de operações vistos nesse capítulo foram o **`Double Dispatch`** e o **`Visitor`**. Esses dois utilizam o recurso de delegar a execução, ou parte dela, para o objeto passado como parâmetro. Dessa forma, o comportamento é determinado pelo parâmetro passado, possibilitando a adição de novas operações a partir da criação de novas abstrações do tipo do parâmetro. Particularmente o **`Visitor`** é um padrão complicado e difícil de ser implementado, porém nas situa-

ções em que ele é aplicável, os ganhos na sua utilização em termos de reúso e manutenção podem ser imensos.

Capítulo 9

Gerenciando Muitos Objetos

"Ninguém é dono da multidão, ainda que creia tê-la dominada."
– Eugene Ionesco

Existem hoje softwares imensos, com milhões de linhas de código! Algumas vezes pergunto às pessoas se elas sabem como alguém consegue trabalhar em uma base de código com milhares de classes e milhões de linhas de código. Quando eu faço isso, alguns coçam a cabeça e tentam imaginar que tipo de mente consegue visualizar, abstrair e trabalhar com tamanha quantidade de informações. A verdade é que, se o software for bem modelado, ninguém precisa conhecer tudo, somente a parte em que está trabalhando e as suas interfaces com o resto.

Porém, mesmo utilizando os padrões que vimos até agora, a quantidade de implementações pode aumentar muito. Isso pode acontecer principalmente quando temos classes granulares, ou quando padrões como `Command`, `Strategy` e `Observer` são utilizados. Nesses padrões, são definidas interfaces que normalmente definem apenas um método, que pode acabar possuindo muitas implementações pequenas. Para os clientes dessas classes, a necessidade de conhecimento de todas as

classes com que ele pode interagir pode se tornar um verdadeiro pesadelo.

Os padrões que serão apresentados nesse capítulo focam em simplificar e tornar escalável o relacionamento entre uma grande quantidade de classes e objetos do sistema. A ideia principal é simplificar a interface entre as classes, tornando mais fácil a interação entre elas. Outras vezes o problema não está exatamente na quantidade de classes, mas na quantidade de objetos que é instanciada. O último padrão desse capítulo mostra como é possível diminuir e controlar essa quantidade de objetos, otimizando a memória dispendida com eles.

9.1 Criando uma Fachada para suas Classes

"A ignorância é uma espécie de bênção."
– John Lennon

É muito comum que uma determinada funcionalidade de um sistema resulte da colaboração de diversos componentes. Por exemplo, ao se concluir uma compra em um sistema de e-commerce, pode ser necessário interagir com componentes responsáveis pelo estoque, pelo controle financeiro e pelo empacotamento e envio de mercadorias. Responsabilizar o cliente desses serviços por interagir com essas classes de forma correta pode causar problemas. Se existirem dois tipos de cliente, por exemplo mobile e web, essa mesma lógica precisará ser replicada nos dois. Isso dificulta mudança nessa lógica e acopla o cliente a diversos componentes.

Quando os padrões são utilizados na criação de uma solução, muitas vezes diversas classes auxiliares vão colaborar. Por exemplo, as classes podem ser encapsuladas com um **`Proxy`**, podem ser compostas por outras classes, ou mesmo receber um **`Visitor`** como parâmetro. Ter o conhecimento de quais classes devem ser criadas e inseridas na composição muitas vezes é complicado para os clientes. Isso não apenas os acopla a implementações específicas, como também à estrutura da solução adotada, dificultando possíveis refatorações futuras.

Em outras palavras, o grande problema aqui é complexidade e acoplamento! Não é desejável que os clientes das classes precisem conhecer todos seus detalhes estruturais ou se acoplar a diversas implementações. O padrão **`Facade`** (pronuncia-se *"façade"* por ser uma palavra de origem francesa) propõe a criação de uma classe intermediária que serve como uma fachada para que o cliente possa acessar as funcionalidades desejadas. Essa classe encapsula a complexidade da interação entre os diversos componentes e desacopla o cliente das implementações.

Estrutura do Facade

A estrutura do **Facade** é bem simples e está representada na Figura 9.1. Deve ser criada uma classe, apresentada na figura como `Fachada`, que recebe as chamadas dos clientes e delega para as classes apropriadas de um subsistema. Por mais que pareça que o **Facade** é um simples redirecionador de chamadas, ele pode ter importantes responsabilidades relativas à coordenação e orquestração dessas chamadas. Dessa forma, as classes clientes irão interagir apenas com a fachada, que os isola da complexidade das classes do subsistema.

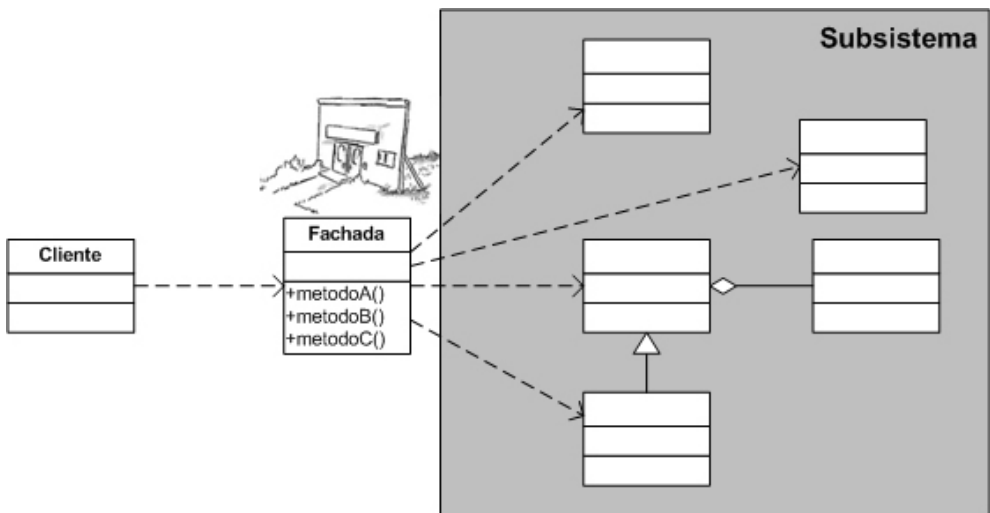



Figura 9.1: Estrutura do Facade

Diversos padrões vistos nesse livro deixam a estrutura de classes mais bem organizada, mais flexível e pronta para evoluir. Porém isso tem um custo de complexidade com que os clientes precisam lidar ao criarem e trabalharem com essa estrutura de classes. Por mais que os padrões de criação vistos nos capítulos 6 e 7 sejam utilizados, os clientes ainda precisam configurar as dependências ou as fábricas. Nesse contexto, o **Facade** encapsula essa complexidade e fornece um método simples que provê aos clientes a funcionalidade desejada.

A partir desse padrão é possível isolar um conjunto de classes do resto da aplicação, deixando a fachada como o único ponto de contato. Com essa estrutura define-se um subsistema que pode evoluir de forma independente e inclusive ser reutilizado em outros contextos. Com essa estrutura modularizada é possível particionar a ar-

quitetura de uma aplicação, organizando os componentes e tornando a manutenção mais simples.

Isolando APIs Externas e Frameworks

Quando utilizamos uma API externa ou um framework, eles precisam ser adaptados às necessidades da aplicação. Por serem mais gerais e com objetivo de atender um grande número de aplicações, a quantidade de configurações que precisam ser feitas e de classes com as quais precisa-se interagir pode ser maior do que o necessário para as necessidades da sua aplicação. Dessa forma, um **`Facade`** pode ser criado para isolar a utilização dessa API ou framework, provendo para a aplicação apenas métodos na medida das suas necessidades. Isso desacopla a aplicação da API utilizada e torna mais simples a interação com aquelas funcionalidades para o resto sistema.

Imagine, por exemplo, uma aplicação que precise acessar um cartão com certificado digital para executar suas funcionalidades. A API que interage com as funcionalidades de criptografia é preparada para lidar com diferentes tipos de cartão e algoritmos criptográficos. Porém, a aplicação geralmente não precisa de toda essa flexibilidade, pois o tipo de cartão que ela irá utilizar já está pré-estabelecido, assim como o algoritmo criptográfico. Dessa forma, utilizando um **`Facade`** para prover os serviços necessários para o resto da aplicação, além de permitir que desenvolvedores possam utilizar a funcionalidade sem conhecer a API, facilita-se a manutenção caso alguma mudança seja necessária nessa parte.

De uma certa forma, a estrutura do **`Facade`** se parece um pouco com a do **`Adapter`**. Uma das diferenças é que o **`Facade`** interage com diversas classes enquanto o **`Adapter`** normalmente encapsula apenas uma classe, apesar de isso não ser uma regra obrigatória. Outra diferença é que o **`Adapter`** normalmente adapta para uma interface conhecida, que o cliente já utiliza por ser implementada por outras classes, e o **`Facade`** define um novo conjunto de métodos com o objetivo de simplificar a interação.

Refatorando para o Facade

A necessidade do **`Facade`** muitas vezes não é percebida no início do projeto.

Isso ocorre porque as interações entre as classes começam de forma simples e objetiva, mas com a evolução do projeto acabam se espalhando para diversas classes e utilizando diversas classes e métodos. Dessa forma, quando o código da aplicação começa a se entrelaçar demais com o de um subsistema e isso passa a dificultar a manutenção, é hora de criar um `Facade` para isolar uma parte da outra.

O passo básico dessa refatoração é simples e está ilustrado na Figura 9.2. O primeiro passo da refatoração é identificar o ponto na classe cliente que faz acesso aos subsistemas e extrair um método com essa lógica. A princípio esse método fica na própria classe, mas o segundo passo da refatoração é justamente movê-lo para o `Facade`. Dessa forma, o acesso ao subsistema ficará concentrado na fachada, isolando-os das classes clientes. Por mais que possa parecer simples, esse procedimento precisa ser repetido para todos os pontos em que o subsistema é acessado.

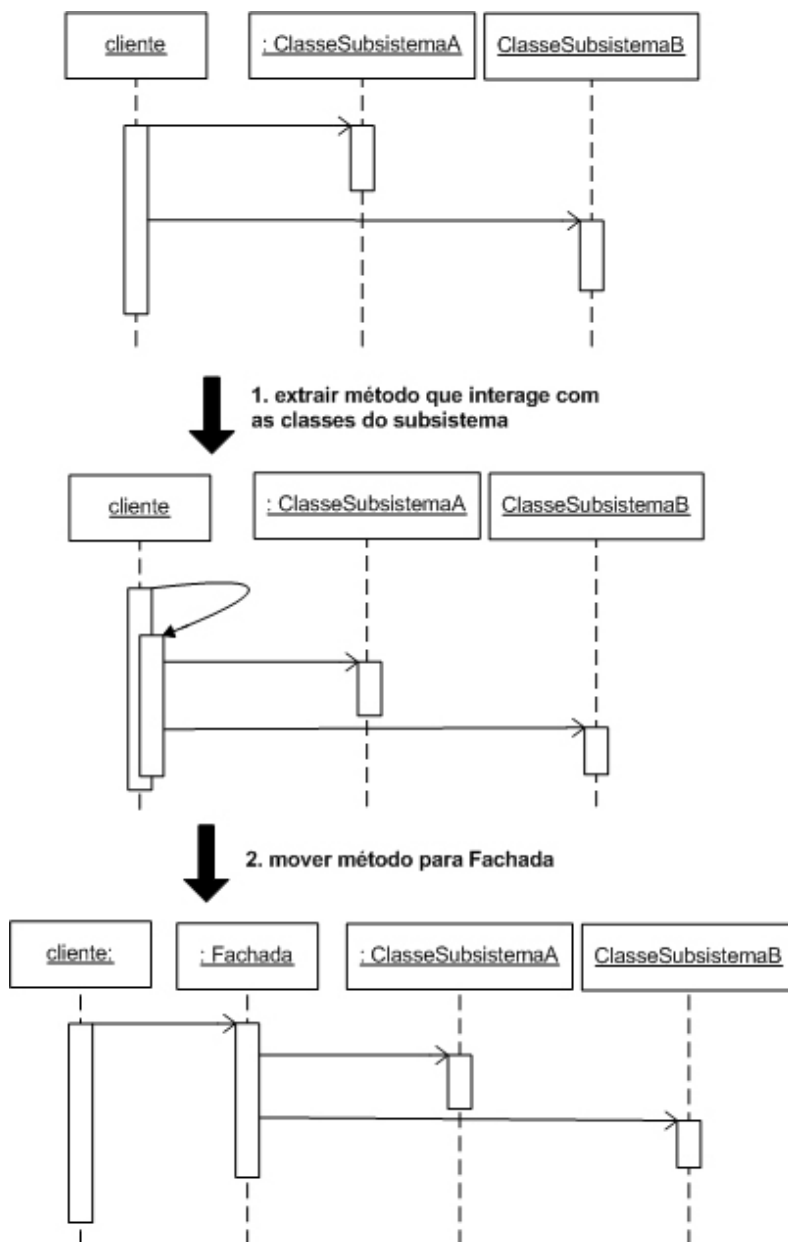



Figura 9.2: Refatorando para a Criação de um Facade

À medida que o código for sendo movido para a fachada, pode ser necessário refatorar esse código, que a princípio estava distribuído entre as classes, e agora está

concentrado em apenas uma. Por exemplo, pode ser que dois métodos movidos para essa classe possuam um código muito parecido. Nesse caso, pode ser que valha a pena introduzir um parâmetro e unir os métodos, ou extrair um método auxiliar comum utilizado pelos dois. A dificuldade da refatoração para a criação de um **Facade** vai depender de quanto o código de acesso ao subsistema está espalhado. Para essa tarefa, vale a pena utilizar alguma ferramenta que detecte as dependências entre as classes, para que possam ser localizados os pontos que dependem do subsistema que está sendo isolado.

Criando uma Fachada para o Gerador de Arquivos

O exemplo do componente gerador de arquivos é um caso no qual foram criadas diversas classes para solucionar o problema de gerar um arquivo a partir de um mapa de propriedades. Para utilizar a solução é preciso saber a subclasse de `GeradorArquivo` correta para definir o tipo de arquivo e compô-la corretamente com os pós-processadores. É claro que com apenas dois tipos de cada, não é tão difícil assim de conhecer as implementações, porém conforme o número de implementações aumentasse, esse componente poderia ficar difícil de utilizar.

Um exemplo de como poderia ser criada uma **Facade** para o gerador de arquivos está apresentado na listagem a seguir. Cada método tem um objetivo bem específico descrito pelo seu nome e, a partir da fachada, as funcionalidades do gerador de arquivo podem ser invocadas diretamente, sem a necessidade de configurações e da criação de classes adicionais. No exemplo, foram criados métodos para a geração das combinações entre os formatos de arquivo e os tipos de pós-processamento, sendo que o caso do pós-processador composto não foi incluído. Dentro do contexto de uma aplicação, seriam incluídos no **Facade** somente os métodos que realmente fossem ser utilizados por ela.

Listagem 9.1 - Exemplo de fachada para gerador de arquivos:

```
public class GeradorArquivoFacade{
    public void gerarXMLCompactado(
            String nome, Map<String,Object> propriedades){
        GeradorArquivo g = new GeradorXML();
        g.setProcessador(new Compactador());
        g.gerarArquivo(nome, propriedades);
    }

    public void gerarPropriedadesCompactado(
```

```
            String nome, Map<String,Object> propriedades){
        GeradorArquivo g = new GeradorPropriedades();
        g.setProcessador(new Compactador());
        g.gerarArquivo(nome, propriedades);
    }

    public void gerarXMLCriptografado(
            String nome, Map<String,Object> propriedades){
        GeradorArquivo g = new GeradorXML();
        g.setProcessador(new Criptografador());
        g.gerarArquivo(nome, propriedades);
    }

    public void gerarPropriedadesCriptografado(
            String nome, Map<String,Object> propriedades){
        GeradorArquivo g = new GeradorPropriedades();
        g.setProcessador(new Criptografador());
        g.gerarArquivo(nome, propriedades);
    }

    //outros métodos
}
```

A ideia de se criar uma fachada para encapsular o acesso a um conjunto de classes não é substituir a API original, mas tornar mais simples o acesso às funcionalidades. No caso do gerador de arquivos, as classes originais continuam existindo e permitindo todo tipo de extensão que era possível anteriormente. Dessa forma, apenas o acesso às funcionalidades foi simplificado, não prejudicando de forma alguma a estrutura e a flexibilidade já existente. A ideia da fachada não é fornecer métodos para cada possibilidade das classes, mas fornecer aquelas necessárias pela aplicação.

Definindo Componentes Plugáveis

O **`Facade`** é um padrão importante para a definição de componentes. A fachada definida é a interface externa do componente, que provê serviços para a aplicação através da coordenação da invocação de suas classes encapsuladas. A definição de uma interface como abstração da fachada permite que existam diversas implementações para aquele componente. Se essa prática for combinada com padrões como **`Dynamic Factory`**, **`Dependency Injection`** ou **`Service Locator`**, é possível que o componente possa ser instanciado dinamicamente. Essa é a base para a definição de uma arquitetura construída com componentes plugáveis, na qual cada um deles pode ser substituído e novos componentes podem ser facilmente adicionados.

9.2 Separando Código Novo de Código Legado

"Deixai-os crescer juntos até a colheita, e, no tempo da colheita, direi aos ceifeiros: ajuntai primeiro o joio, atai-o em feixes para ser queimado; mas o trigo, recolhei-o no meu celeiro."
– Mateus 13:30

Em seu livro sobre Domain-driven Design [11], Eric Evans apresenta um padrão chamado **`Anti-Corruption Layer`**. Esse padrão é aplicável em cenários onde é necessário acessar código legado, onde não necessariamente o modelo de objetos é bem construído e compatível com o da nova aplicação. Esse código legado pode ser um sistema completo, um componente ou até mesmo um conjunto de classes. Um dos riscos nesse caso é de o modelo antigo "contaminar" a nova aplicação, a qual tenta se adaptar ao paradigma e à modelagem do código antigo. Por outro lado, essa integração muitas vezes é importante, pois pode proporcionar reaproveitamento de código e gerar um grande valor para o cliente.

O **`Anti-Corruption Layer`** propõe a criação de uma camada entre a nova aplicação e o código legado. Ela é responsável por traduzir as chamadas feitas pela aplicação em invocações para classes do sistema legado. A partir dessa nova camada, é possível abstrair a existência do código legado do resto da aplicação, oferecendo os serviços de forma consistente com o modelo e a modelagem da nova aplicação.

Na implementação desse padrão, o **`Facade`** é normalmente utilizado para fornecer os serviços necessários do código legado, encapsulando todo acesso a ele. Dentro dessa fachada pode ser necessária utilização de classes responsáveis por traduzir as entidades do modelo de um dos sistema para o outro. Isso pode ocorrer tanto com as classes da nova aplicação que serão recebidas como parâmetro quanto com as classes retornadas pelo sistema legado.

A tradução entre duas classes é possível quando o importante forem os dados que ela carrega. Porém, quando a classe retornada possuir métodos que precisam ser invocados, não faz sentido duplicá-los na classe do novo sistema. Nesse caso, pode ser utilizado um **`Adapter`** que encapsula uma classe legada para um interface utilizada pelo novo sistema. Dessa forma, existirá um método no adaptador que irá fazer os ajustes necessários para delegar a funcionalidade para o método da classe legada.

9.3 Mediando a Interação entre Objetos

"Cada um de nós escolherá o mediador que seleciona a informação segundo os critérios que preferimos."
– F. Sarsfield Cabral

O **`Facade`** é um padrão que pode ser utilizado para o encapsulamento das funcionalidades fornecidas por um subsistema. Essas funcionalidades são disponibilizadas através de uma classe e consumidas por objetos e componentes da aplicação. Porém, nem sempre a interação entre os objetos de um software acontece em apenas uma direção. Os objetos podem se relacionar com outros de formas complexas, inclusive de forma a haver uma comunicação bidirecional entre eles. Por mais que essa interação possa acontecer de forma bem definida, podem ser criadas interdependências desestruturadas e difíceis de compreender.

Um contexto em que a lógica de interação entre componentes costuma ser tornar complexa é em interfaces gráficas. Por exemplo, a escolha de um valor em um combo box pode ser o gatilho para o recarregamento de uma lista e ao mesmo tempo fazer com que certos campos sejam desabilitados. Em dois campos onde precisam ser entradas a senha e sua verificação, a mudança do valor em um dos campos pode disparar uma lógica de validação que faz a comparação com o outro. Quando um componente é ligado diretamente com outros e as interações são complexas, fica muito difícil entender e dar manutenção nesses relacionamentos, que precisam ser criados um a um, em uma relação muitos-para-muitos.

É justamente para esse tipo de cenário que o **Mediator** deve ser utilizado. Esse padrão propõe a criação de uma classe que serve como mediadora entre os objetos. Dessa forma, ao invés dos objetos enviarem requisições e receberem requisições de vários outros, ela irá interagir apenas com o mediador. Essa classe será responsável por receber as requisições dos objetos e enviá-las para os objetos que as devem receber.

Aumentando o Número de Observadores e Observados

Para exemplificar a utilização desse padrão, vamos retomar o exemplo apresentado no Capítulo 3 para o padrão **Observer**. Nesse exemplo, a classe `CarteiraAcoes` representa uma carteira de ações que pode possuir observadores, sendo que o método `addObservador()` é utilizado para a adição de um observador. Cada observador é uma classe que implementa a interface `Observador`, que recebe uma chamada no método `mudancaQuantidade()` quando ações são adicionadas ou removidas da carteira.

Dentro desse cenário, as coisas podem se complicar quando o número de carteiras e de observadores puder ser grande. Por exemplo, se houver dez carteiras de ações a serem observadas por dez diferentes observadores em um determinado contexto, serão necessárias cem invocações de método para adicionar cada um dos observadores em cada uma das carteiras. Essa ligação direta entre os objetos pode tornar difícil o gerenciamento de quais observadores estão recebendo eventos de quais carteiras.

A utilização de um **Mediator** é uma alternativa para gerenciar a relação entre os observadores e as carteiras de ação. A classe `GrupoObservacao`, cuja listagem está apresentada a seguir, é uma mediadora que representa um grupo de carteiras que podem ser observadas por um grupo de observadores. Para cumprir esse objetivo, essa classe implementa a interface `Observador` e se adiciona como observadora em todas as carteiras adicionadas a ela. Além disso, ela também recebe observadores, os quais são adicionados em uma lista mantida internamente. Assim, quando alguma das carteiras notificar o mediador de algum evento, esse será repassado a todos os observadores.

Listagem 9.2 - Mediador para interação entre a carteira de ações e seus observadores:

```
public class GrupoObservacao implements Observador {
    private List<Observador> obs = new ArrayList<>();
```

```
    public void mudancaQuantidade(String acao, Integer qtd) {
        for(Observador o: obs)
            o.mudancaQuantidade(acao, qtd);
    }
    public void addObservador(Observador o) {
        obs.add(o);
    }
    public void addCarteira(CarteiraAcoes ca) {
        ca.addObservador(this);
    }
}
```

Cada observador será adicionado apenas na classe `GrupoObservacao`, que se encarregará de repassar os eventos gerados por todas as carteiras. De forma similar, cada carteira também terá apenas um observador, que será o mediador. Portanto, essa classe age como intermediária para a comunicação entre esses objetos.

O exemplo apresentado mostra um caso simples de **`Mediator`**, porém imagine que esse cenário se complique um pouco mais. Considere, por exemplo, que determinados observadores estejam interessados apenas em eventos a partir de uma certa quantidade ou somente de um grupo de ações. Esse padrão cria um contexto de interação entre os objetos onde esse tipo de regra pode ser introduzida e mais facilmente gerenciada.

Estrutura do Mediator

A estrutura do padrão **`Mediator`** é bem ilustrada pela metáfora utilizada em seu nome. Existe uma classe mediadora que serve como intermediária para interação entre os objetos da aplicação, como mostra a Figura 9.3. Nessa estrutura, a classe representada como `Mediador` recebe chamadas de `RealizadorChamadas` e, de acordo com a lógica de interação entre os objetos, invoca métodos nas classes do tipo `RecebedorChamadas`. A classe `RealizarRecebedor`, representada no diagrama, ilustra o caso em que uma mesma classe recebe e realiza invocações no mediador.

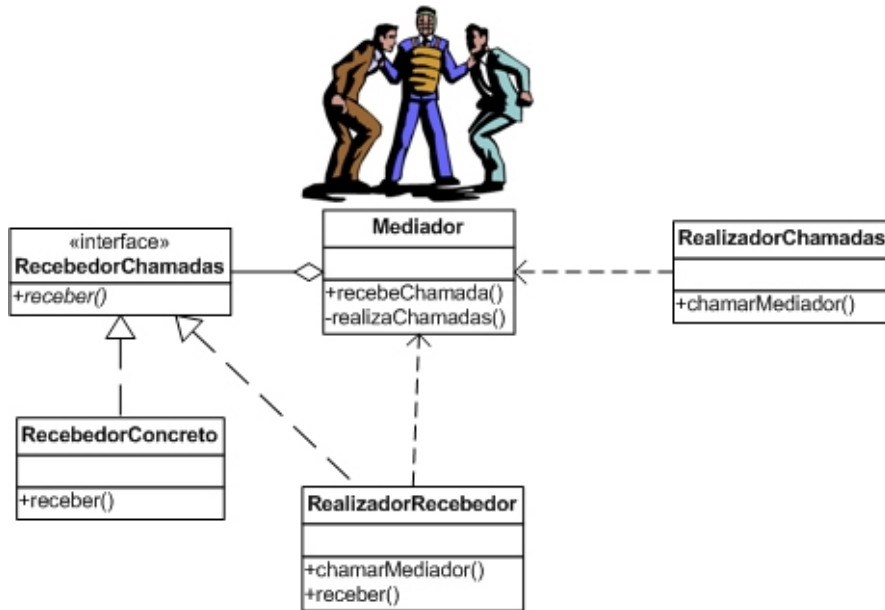



Figura 9.3: Estrutura do padrão Mediator

Os objetos que são invocados pelo mediador, por precisarem ser acessíveis a ele, normalmente são armazenados em atributos internos. Já as classes que invocam funcionalidades nele precisam, de alguma forma, obter sua referência. Para isso, podem ser aplicadas as estratégias de criação de objetos apresentadas em capítulos anteriores. Por exemplo, se houver apenas um mediador para toda a aplicação, um **`Singleton`** pode ser utilizado para que as classes tenham uma forma centralizada de acessar a sua instância. Uma outra alternativa, seria o uso de **`Dependency Injection`**, de forma que o mediador seria injetado nas classes que precisassem se comunicar com ele.

Um dos principais benefícios desse padrão é o desacoplamento entre as classes que precisam realizar uma chamada e as que devem tratá-las. Como o mediador é colocado entre as classes, em nenhum momento existe um acesso direto entre elas. O **`Mediator`** centraliza o gerenciamento do relacionamento entre objetos, o que simplifica a troca das instâncias e a adição de novas no contexto de interação. Em outras palavras, caso um novo objeto precise interagir com outros, em vez dele ser ligado a cada objeto, ele só precisa estar ligado ao mediador.

O **`Mediator`** é indicado quando as relações entre os objetos forem complexas

o suficiente, de forma a valer a pena que essa responsabilidade seja concentrada em uma classe. Um dos motivos para essa complexidade seria uma grande quantidade de objetos, os quais precisariam estar ligados uns aos outros. Outro motivo seriam regras para essa interação entre os objetos, como condições para que um determinado objeto receba uma chamada. Com o uso desse padrão, essas regras migrariam das classes para o mediador. Por exemplo, ao invés de uma classe filtrar com quais objetos ela deve interagir ou quais chamadas ela deve tratar, essa responsabilidade seria movida para o mediador. É importante ressaltar que apesar de as relações tratadas pelo mediador serem complexas, elas precisam seguir regras bem definidas.

O passo básico para a refatoração de adição de um mediador em um conjunto de classes não é muito diferente do apresentado na Figura 9.2, para a adição de um **Facade**. A lógica de interação entre os objetos precisaria ir sendo extraída em métodos, os quais seriam movidos para o mediador em seguida. Porém, à medida que novas interações fossem sendo adicionadas, o mediador precisaria ir incorporando as novas regras, sem eliminar as antigas. Uma prática que facilita essa transição são os mediadores implementarem as abstrações das classes que recebem as chamadas, pois assim ele poderá ser invocado de forma transparente pelas classes já existentes. Isso foi feito no exemplo da classe `GrupoObservacao`, que implementou a interface `Observador` para poder ser facilmente adicionada na classe `CarteiraAcoes` já existente.

Lançando Eventos no Spring

Uma forma interessante de se utilizar um **Mediator** é através de eventos. Todas as requisições são passadas para o mediador na forma de um evento, o qual é tratado pelas classes interessadas naquele tipo de evento. O framework Spring, apresentado no Capítulo 7 para o uso de injeção de dependências, possui uma funcionalidade que permite o lançamento e o tratamento de eventos pelas classes declaradas no seu contexto.

Essa seção irá apresentar como o exemplo da carteira de ações seria implementado utilizando o Spring como mediador. A primeira classe que precisa ser definida é a que representa o evento que será lançado. Essa classe precisa conter as informações que a classe que cria o evento deseja comunicar para a que o receberá. A listagem a seguir apresenta a classe `MovAcao`, que representa um evento de movimentação de uma ação. Essa classe precisa estender a classe `ApplicationEvent`, definida pelo próprio Spring.

Listagem 9.3 - Definição de um novo tipo de evento:

```
public class MovAcao extends ApplicationEvent {
    private String acao;
    private Integer qtd

       public MovAcao(String acao, Integer qtd) {
        this.acao = acao;
        this.qtd = qtd;
       }
       public String getAcao() {
           return acao;
       }
       public Integer getQtd() {
           return qtd;
       }
}
```

Em seguida é preciso definir a classe que irá gerar os eventos e notificar o Spring. No exemplo apresentado, esse seria o caso da classe `CarteiraAcoes`, apresentada na listagem a seguir, que deve gerar um `MovAcao` toda vez que a quantidade de uma ação for alterada através do método `adicionaAcoes()`. Para que ela possa gerar um evento, ela precisa receber do Spring uma injeção de dependências de uma instância de `ApplicationEventPublisher`, cujo método `publish()` deve ser chamado a cada geração de evento. Nesse caso, o framework utiliza a estratégia de **`Dependency Injection`** por meio de interfaces, na qual a classe precisa implementar a interface `ApplicationEventPublisherAware` para receber a dependência no método setter definido por ela.

Listagem 9.4 - Classe que publica os eventos no Spring:

```
public class CarteiraAcoes implements ApplicationEventPublisherAware {
    private Map<String,Integer> acoes = new HashMap<>();
    private ApplicationEventPublisher publicador;

       public void
           setApplicationEventPublisher(ApplicationEventPublisher p) {

          this.publicador = p;

       }
```

```
    public void adicionaAcoes(String acao, Integer qtd) {
        if(acoes.containsKey(acao)){
            qtd += acoes.get(acao);
        }
        acoes.put(acao, qtd);
        MovAcao evento = new MovAcao(acao,qtd)
        publicador.publishEvent(evento)
    }
}
```

A classe que estiver interessada em um evento deve implementar a interface `ApplicationListener`, com o tipo genérico igual à classe do evento que se deseja receber. Dessa forma, toda vez que um evento daquele tipo for lançado, o Spring irá invocar o método `onApplicationEvent()` naquela classe. A listagem a seguir mostra o exemplo da classe `AcoesLogger` que escreve no console as informações das instâncias de `MovAcao` recebidas.

Listagem 9.5 - Classe que recebe os eventos do tipo MovAcao do Spring:

```
public class AcoesLogger implements ApplicationListener<MovAcao>{
    public void onApplicationEvent(MovAcao evento) {
          System.out.println("Alterada a quantidade da ação "
                + evento.getAcao() + " para " + evento.getQtd());
       }
}
```

Para juntar os pedaços e colocar o Spring como mediador das classes, é preciso criar um arquivo de configuração com a definição dessas classes. No exemplo, esse arquivo se chama `beans.xml` e seu conteúdo está apresentado na listagem a seguir.

Listagem 9.6 - Arquivo beans.xml para configuração dos beans do Spring:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
   <bean id="carteira" class="br.com.casadocodigo.CarteiraAcoes"/>
   <bean id="logger" class="br.com.casadocodigo.AcoesLogger"/>
</beans>
```

Finalmente, para ilustrar como esse código seria utilizado, a próxima listagem mostra um trecho de código que utiliza as classes criadas. Nesse código é criado um

contexto do Spring através do arquivo XML apresentado e em seguida a instância da classe `CarteiraAcoes` é recuperada. Quando o método `adicionaAcoes()` é invocado, o evento será gerado e recebido pela classe `AcoesLogger`.

Listagem 9.7 - Código que cria o contexto da aplicação e provoca geração de eventos:

```
ConfigurableApplicationContext contexto =
    new ClassPathXmlApplicationContext("beans.xml");
CarteiraAcoes c = (CarteiraAcoes) contexto.getBean("carteira");
c.adicionaAcoes("ABC", 300);
c.adicionaAcoes("XYPLZ", 700);
```

Essa abordagem baseada em eventos é uma forma interessante de implementação de mediadores. Os eventos são lançados pelas classes para o Spring sem que elas saibam que outras classes se interessam por eles. Da mesma forma, os eventos são tratados por classes que não têm conhecimento de onde foram originados. Essa é uma forma de desacoplar os componentes de um sistema, sem impedir que eles se comuniquem com os outros.

9.4 Reaproveitando Instâncias com Flyweight

"Uma classe deve ser imutável a não ser que tenha uma boa razão para fazê-la mutável."
– Joshua Bloch

As seções anteriores abordaram padrões que visam simplificar o relacionamento e a comunicação entre as classes de um sistema. Tanto o **`Facade`** quanto o **`Mediator`** simplificam as interfaces, tornando a estrutura do software mais modular. Diferentemente, o **`Flyweight`** ataca cenários onde existe a criação de muitos objetos de uma mesma classe, que muitas vezes acabam sendo semelhantes. A solução desse padrão consiste no reaproveitamento da mesma instância para representação de objetos semelhantes, sendo que o que irá diferenciá-la é o contexto no qual ela será inserida.

O clássico exemplo desse padrão utilizado no GoF [10] é o de um editor de texto que precisa representar as letras de um documento. Se cada letra do documento fosse representada por um objeto diferente, a quantidade de objetos seria muito grande. Imagine, por exemplo, quantas vezes a letra "a" aparece em um texto. Utilizando

o **Flyweight**, cada letra seria representada por apenas uma instância, que seria utilizada sempre que ela fosse necessária. O que diferenciaria, por exemplo, as diferentes letras "a" do texto, seriam o local e o contexto em que estariam inseridas. Dessa forma, propriedades que possam variar, como o tamanho e tipo da fonte, não seriam propriedades da letra e sim do trecho de texto em que estaria inserida.

Por mais que esse exemplo do editor de texto seja bem ilustrativo para o entendimento do padrão, acredito que muito poucos desenvolvedores nos dias de hoje trabalhem com editores de texto. Adicionalmente, a representação de caracteres em Strings e texto é um problema já bem resolvido pelas bibliotecas nativas de linguagens de programação mais modernas, como o Java. Então será que esse padrão ainda é aplicável em sistemas atuais? Em que outros cenários ele poderia ser utilizado? É justamente sobre isso que se trata essa seção!

Status de Itens de um Pedido em um Sistema de E-commerce

Imagine que em um sistema de e-commerce seja necessário controlar o status de cada item de um pedido. Cada item da compra é composto com uma instância da classe `StatusItem` apresentada na listagem a seguir. Além do nome do status, essa classe também possui algumas propriedades booleanas que são utilizadas na lógica do sistema para verificar se algumas ações podem ser tomadas pelo usuário. Quando o status de um item é alterado, uma nova instância de `StatusItem` referente ao novo status é criada e inserida na instância.

Listagem 9.8 - Classe que representa o status de um item de uma compra:

```
public class StatusItem {
    private String nome;
    private boolean podeCancelar;
    private boolean compraConcluida;

    public StatusItem(String nome, boolean podeCancelar,
            boolean compraConcluida){
        this.nome = nome;
        this.podeCancelar = podeCancelar;
        this.compraConcluida = compraConcluida;
    }
    public String getNome(){
        return nome;
    }
```

```
    public boolean isCompraConcluida(){
        return compraConcluida;
    }
    public boolean podeCancelar(){
        return podeCancelar;
    }
}
```

A perceber que a aplicação estava ocupando um grande espaço em memória, foi utilizada uma ferramenta de *profiling* para investigar quais instâncias estavam ocupando a memória da máquina virtual. Nessa análise, uma das surpresas foi a presença de um número muito grande de objetos da classe `StatusItem`. Mesmo sendo uma classe pequena, a imensa quantidade de instâncias ocupava uma parte significativa da memória. Por ser uma aplicação em que diversos usuários acessam de forma simultânea, e pelo fato de cada ordem de compra poder possuir diversos itens, o número de `StatusItem` na memória em um determinado momento é sempre alto. Porém, por diversas instâncias possuírem as mesmas informações, esse ponto foi um dos escolhidos para otimização.

O QUE É UMA FERRAMENTA DE PROFILING?

Uma ferramenta de *profiling* é um software utilizado por desenvolvedores para fazer análise de desempenho de aplicações. Ferramentas de *profiling* para Java normalmente se conectam à máquina virtual que está executando a aplicação e colhem diversas informações, como o tempo de execução dos métodos e a quantidade de objetos em memória, entre outras coisas. Sendo assim, é possível saber que método está demorando mais tempo para executar ou qual é a classe cujas instâncias estão ocupando mais memória. Se em uma análise dessas for constatado que existem muitas instâncias semelhantes de uma mesma classe na memória, o **`Flyweight`** é certamente um padrão candidato para diminuir significativamente esse número.

Apresentando o Flyweight

O **`Flyweight`**, cuja tradução seria algo como peso-mosca ou peso-pena, é um padrão cujo objetivo é permitir a representação de um número grande de objetos

de forma eficiente. Ele é aplicável em cenários onde existem diversas instâncias da mesma classe em memória que são similares. Dessa forma, a solução proposta pelo padrão é reutilizar a mesma instância em todos os locais onde objetos semelhantes precisarem ser utilizados.

A Figura 9.4 apresenta a estrutura do padrão. A classe apresentada com o nome `FabricaFlyweight` é responsável por retornar as instâncias de `Flyweight` para as classes clientes. Normalmente a fábrica utiliza uma chave para identificar a instância que deve ser retornada, que normalmente é uma string ou um número que identifica logicamente dentro da aplicação o objeto desejado. A fábrica pode inicializar os objetos de forma "preguiçosa", criando-os na primeira vez que forem solicitados, armazenando-os internamente e retornando a mesma instância em solicitações posteriores. Ou, como alternativa, as instâncias podem ser criadas logo na inicialização da própria fábrica, o que é indicado quando houver poucas instâncias.

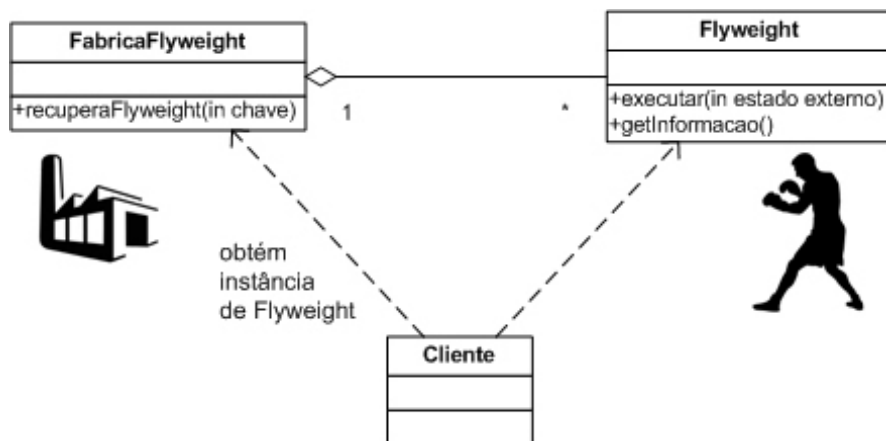



Figura 9.4: Funcionamento do Flyweight

Uma característica muito importante do **`Flyweight`** é que os objetos representados por ele precisam ser imutáveis. Em outras palavras, depois de criados, o estado interno desses objetos não pode ser mais alterado. O principal motivo para isso é que a mesma instância estará sendo utilizada em diversos contextos da aplicação, e se ela for de alguma forma modificada, todos os outros contextos serão afetados. Mais à frente nesse mesmo capítulo serão apresentados cuidados que se deve ter em Java para a criação de classes imutáveis.

Um outra característica das classes **`Flyweight`**, justamente por sua imutabi-

lidade, está no fato de seus métodos algumas vezes dependerem do estado externo para sua execução. Toda informação que precisar ser alterada ou que for dependente do estado da execução não pode ser armazenada dentro do **Flyweight**. Por isso mesmo, se essa classe precisar executar alguma lógica que depende do contexto onde está, as informações precisam ser passadas como parâmetro. Na Figura 9.4, esse tipo de método está representado pelo método `executar()` na classe `Flyweight`.

A principal consequência desse padrão é a economia de memória que se pode ter com a reutilização de instâncias em diversos pontos da aplicação. Em aplicações que tratam requisições de diversos usuários de forma simultânea, como em aplicações Web, esse padrão pode ter um grande impacto para a representação de objetos muito utilizados. A melhora de desempenho também pode ter impacto em processamento, visto que diversas instâncias não precisarão mais ser criadas, e posteriormente coletadas pelo Garbage Collector. Além disso, se a aplicação realmente garantir que existe apenas uma instância para cada tipo de objeto daquela classe, as comparações entre eles podem ser feitas utilizando o operador "==", que é muito mais eficiente que o método `equals()`.

Compartilhando as Instâncias que Representam o Status

Voltando ao problema apresentado anteriormente, as instâncias da classe `StatusItem` são boas candidatas à implementação do padrão **Flyweight**. Isso porque são objetos imutáveis no contexto da aplicação, ou seja, um item irá mudar de status, mas não irá modificar as informações de um determinado status. Isso permite que as instâncias dessa classe possam ser compartilhadas por diversos objetos. O fato de haver um número muito grande de instâncias similares dessa classe na aplicação também motiva a implementação do padrão.

O primeiro passo para implementá-lo é a criação de uma fábrica responsável pela criação e retorno dos objetos. No caso foi criada a classe `FabricaStatusItem`, representada na listagem a seguir. Nesse contexto, deve haver apenas uma instância dessa fábrica para que os objetos de `StatusItem` sejam armazenados em apenas um local. Para isso, essa classe implementa o padrão `Singleton`, armazenando a instância no atributo estático `instance` e retornando-o através do método `getInstance()`.

Listagem 9.9 - Fábrica para obtenção das instâncias de StatusItem:

```
public class FabricaStatusItem {
    private static FabricaStatusItem instance =
```

```
        new FabricaStatusItem();

    public static FabricaStatusItem getInstance() {
        return instance;
    }

    private Map<String,StatusItem> mapa;

    private FabricaStatusItem(){
        mapa = new HashMap<>();
        mapa.put("CARRINHO", new StatusItem(
            "CARRINHO", true, false));
        mapa.put("FECHADO", new StatusItem(
            "FECHADO", true, false));
        mapa.put("PAGO", new StatusItem(
            "PAGO", true, true));
        mapa.put("ENVIADO", new StatusItem(
            "ENVIADO", false, true));
        mapa.put("ENTREGUE", new StatusItem(
            "ENTREGUE", false, true));
    }
    public StatusItem get(String nome){
        if(!mapa.containsKey(nome))
            throw new RuntimeException("Status inexistente:"+nome);
        return mapa.get(nome);
    }
}
```

A classe `FabricaStatusItem` cria todos os `StatusItem` no construtor e os armazena em uma mapa. Essa é uma estratégia válida pois o número de objetos é pequeno. Outra possibilidade seria ir criando os objetos à medida que forem solicitados. Esse caso é interessante, por exemplo, quando os objetos são armazenados em uma base de dados. Assim somente os objetos que realmente forem utilizados serão mantidos em memória.

Além da criação da fábrica, é preciso substituir os locais onde a classe `StatusItem` é instanciada pela invocação da fábrica. Uma forma de ajudar a detectar esses pontos é declarar o construtor da classe com o modificar de acesso *default*, e colocar somente a fábrica no mesmo pacote que ele. Isso gerará um erro de compilação em todas as classes fora desse pacote que tentarem utilizá-lo.

Cuidados para Criar Objetos Imutáveis e Únicos

Criar classes que geram objetos imutáveis pode não ser algo tão trivial quanto se imagina. Principalmente em linguagens orientadas a objetos em que as classes podem ser estendidas e comportamentos podem ser sobrepostos. Na implementação do **`Flyweight`** é importante ter certeza de que as instâncias compartilhadas realmente não permitem modificação, pois isso poderia resultar em um problema crítico e difícil de detectar. Abaixo estão algumas diretivas a serem seguidas para a implementação de classes imutáveis:

- Não adicione métodos setter ou outros métodos que modifiquem os atributos da classe. Isso irá impedir que externamente os clientes do objeto o modifiquem.

- Crie todos os atributos como `private` e `final`. Isso irá impedir que eles sejam inacessíveis a subclasses e que possam ser modificados após a sua primeira atribuição de valor.

- Impeça a criação de subclasses. Isso pode ser feito de forma direta declarando a classe como `final`. Outra alternativa é declarar o construtor como `private` e utilizar um **`Static Factory Method`** para a criação das instâncias. Isso é importante para impedir que subclasses adicionem novas características mutáveis.

- Não retorne diretamente atributos com objetos mutáveis. Em outras palavras, se você retornar um objeto mutável armazenado em um atributo, ele poderá ser modificado pelos clientes, mesmo estando definido como `final`. Caso seja mesmo necessário retorná-lo, crie uma cópia e o retorne.

- Cuidado ao receber objetos mutáveis na construção do objeto, pois eles podem ser modificados pelos clientes posteriormente, mesmo não sendo retornados pela classe. O que recomenda-se nesse caso é a criação de uma cópia do objeto mutável recebido para o armazenamento em um atributo do objeto imutável.

Analisando a classe `StatusItem` apresentada anteriormente, é possível observar que ela não segue todas as recomendações para a criação de objetos imutáveis. Apesar de não possuir métodos setter para modificações no objeto, os atributos não possuem o modificador `final` e a classe pode ser estendida. A listagem a seguir mostra a classe `StatusItem` modificada segundo as recomendações para objetos

imutáveis. Observe que o construtor foi definido como privado e foi definido um **Static Factory Method** para impedir que sejam definidas subclasses. Como nenhum dos atributos dessa classe é mutável, não foi preciso se preocupar com o retorno e com o recebimento deles como parâmetro.

Listagem 9.10 - Tornando a classe StatusItem imutável:

```
public class StatusItem {
    private final String nome;
    private final boolean podeCancelar;
    private final boolean compraConcluida;

    static void StatusItem criar(String nome,
            boolean podeCancelar, boolean compraConcluida) {
        return new StatusItem(nome, podeCancelar, compraConcluida);
    }

    private StatusItem(String nome, boolean podeCancelar,
            boolean compraConcluida) {
        this.nome = nome;
        this.podeCancelar = podeCancelar;
        this.compraConcluida = compraConcluida;
    }
    //métodos de acesso as propriedades
}
```

Um outro cuidado que é preciso ter seria em relação à unicidade dos objetos do **Flyweight**. Além da existência da fábrica, é preciso ter certeza de que ela está sendo invocada em todos os locais onde o objeto é criado. Na classe `StatusItem` dessa seção, o **Static Factory Method** chamado `criar()` é definido com o modificador de acesso *default* para que somente a fábrica, localizada no mesmo pacote que ele, possa utilizá-lo.

Porém, somente isso pode não ser suficiente para garantir essa unicidade. Um caso que precisa de um tratamento especial é quando a classe que contém o **Flyweight** precisa ser serializada. Normalmente isso ocorre quando ela é enviada remotamente como parâmetro ou quando é persistida em um arquivo. Nesse caso, no momento da desserialização uma nova instância do **Flyweight** será criada. Nesses casos, é preciso sobrepor o mecanismo de serialização da classe que contém a instância do **Flyweight** para que ela seja recuperada da fábrica quando for des-

serializada.

Para exemplificar esse caso, será considerada a classe `Item` que é composta pela classe `StatusItem`, e no exemplo precisa ser serializável pois é enviada remotamente como parâmetro para um EJB. Para evitar que um atributo do tipo `StatusItem` seja incluído de forma inadequada em uma classe serializável, ela deve ser mantida sem a implementação da interface `Serializable`. Dessa forma, caso haja a tentativa de serializá-la, mesmo que dentro de uma outra, ocorrerá um erro.

Para que seja possível serializar uma classe composta por um `StatusItem`, é necessário declarar o atributo como `transient` para que ele não seja incluído no algoritmo de serialização. Em seguida é preciso customizar o algoritmo de serialização para que ele de alguma forma guarde o status do objeto e o recupere da fábrica depois. Isso pode ser feito criando os métodos privados `writeObject()` e `readObject()` na classe. No método `writeObject()`, o método `defaultWriteObject()` é chamado para que o algoritmo de serialização padrão seja executado e em seguida o nome do status, que é utilizado para a recuperação da fábrica, também é armazenado. De forma similar, no método `readObject()`, o método `defaultReadObject()` é chamado inicialmente e em seguida é lida a string com o nome do status. Essa string é utilizada para recuperar o `StatusItem` da fábrica e o atribuir ao atributo da classe.

Listagem 9.11 - Sobrepondo o mecanismo de serialização da classe Item:

```
public class Item implements Serializable {
    private transient StatusItem status;

    //outras propriedades
    //métodos getters e setters omitidos

    private void writeObject(ObjectOutputStream oos)
                                throws IOException {
        oos.defaultWriteObject();
        oos.writeObject(status.getNome());
    }
    private void readObject(ObjectInputStream ois)
            throws ClassNotFoundException, IOException {
        ois.defaultReadObject();
        String status = ois.readObject().toString();
        this.status = FabricaStatusItem.getInstance().get(status);
    }
```

```
}
```

O código da listagem a seguir pode ser utilizado para testar se o mecanismo de serialização está funcionando corretamente. O método chamado `copiarPorSerializacao()` cria uma cópia do objeto a partir de sua serialização e desserialização. Dessa forma, é possível criar um objeto da classe `Item`, criar uma cópia e em seguida verificar se a instância configurada na variável transiente, que no caso representa o **Flyweight**, é a mesma.

Listagem 9.12 - Sobrepondo o mecanismo de serialização da classe Item:

```
public class TesteSerialização {
    public static void main(String[] args) throws Exception {
        Item i1 = new Item();
        i1.setProduto("Livro Design Patterns");
        i1.setStatus(FabricaStatusItem.getInstance().get("PAGO"));
        //seta outras propriedades

        Item i2 = (Item) copiarPorSerializacao(i1);
        if(i1.getStatus()==i2.getStatus()){
            System.out.println("Mesma instância!");
        }
    }
    public static Serializable copiarPorSerializacao(Serializable obj)
            throws Exception {
        ObjectOutputStream output = null;
        ObjectInputStream input = null;
        try {
            ByteArrayOutputStream byteOutput =
                    new ByteArrayOutputStream();
            output = new ObjectOutputStream(byteOutput);
            output.writeObject(obj);
            output.flush();
            byte[] byteArray = byteOutput.toByteArray();
            ByteArrayInputStream byteInput =
                    new ByteArrayInputStream(byteArray);
            input = new ObjectInputStream(byteInput);
            return (Serializable) input.readObject();
        } finally {
            output.close();
            input.close();
        }
```

```
    }
}
```

O mesmo cuidado que se tem com a serialização do objeto, também é preciso ter com qualquer tipo de persistência. Se a sua aplicação utiliza JPA, por exemplo, se a classe `Item` for persistente, deve-se tomar cuidado para que instâncias de `StatusItem` não sejam criadas de forma automática pelo framework de persistência. A solução para esse problema é similar à apresentada para a serialização. O atributo deve ser marcado com a anotação `@Transient` e métodos com anotações do tipo `@PrePersist` e `@PostLoad` devem ser criados para respectivamente persistir a chave em um outro atributo e recuperar o objeto da fábrica depois do carregamento dos dados no objeto.

Como o Flyweight é Usado Para Strings em Java

A classe `String` em Java é um objeto imutável. Todos os métodos que modificam uma string, como para concatenação ou extração de substring, retornam uma nova instância de `String` e não modificam a original. Dessa forma, a cadeia de caracteres utilizada internamente pela `String` pode ser compartilhada por diversas instâncias dessa classe. Isso fica bem claro em métodos como `substring()`, em que a mesma cadeia é utilizada na nova instância de `String` retornada. Isso seria um exemplo do uso do **`Flyweight`** para a economia de memória na máquina virtual.

9.5 Considerações Finais do Capítulo

Esse capítulo tratou de questões relacionadas ao gerenciamento do relacionamento entre os objetos de uma aplicação. Para entender uma classe, é preciso entender não apenas seus elementos, mas também as interfaces com as classes com que se relaciona. Quando o número de classes aumenta, é essencial que elas sejam organizadas e que se busque simplificar a interface entre elas. Quando o número de instâncias é muito grande, é importante gerenciar o relacionamento entre elas para que seja possível entender o estado do sistema em um determinado momento.

O padrão **`Facade`** foi apresentado como uma forma de dividir a aplicação em subsistemas. Ele cria uma interface única que coordena a invocação de diversas clas-

ses para fornecer serviços ao resto da aplicação. Como consequências principais, as classes encapsuladas ficam desacopladas do resto da aplicação, e a interface para obter sua funcionalidade é simplificada. De forma similar, o **`Mediator`** serve como um intermediário para relação entre objetos do sistema. Por centralizar e gerenciar os relacionamentos, o mediador retira das classes essa responsabilidade que fica espalhada entre elas e desacopla a classe que gera uma chamada de método da classe que a recebe.

Em seguida, focando na diminuição do número de instâncias similares de uma mesma classe, o **`Flyweight`** propõe que essas instâncias sejam reaproveitadas em diversos pontos da aplicação. Para isso, é preciso criar uma fábrica desses objetos, a qual retorne a mesma instância quando forem necessários objetos similares. Também é preciso que essa classe compartilhada seja imutável, para que alterações realizadas em um contexto não afetem outros em que a instância estiver sendo reutilizada. Além do padrão, também foram apresentadas questões mais específicas da linguagem Java para a implementação desse padrão na prática.

Capítulo 10

Indo Além do Básico

"Ao infinito... E além!"
– Buzz Lightyear, Toy Story

Durante todo esse livro trilhamos um caminho que nos apresentou diversos padrões que podem ser utilizados para projetar um software orientado a objetos. Cada capítulo focou em uma técnica ou em tipo de problema, para os quais alguns padrões foram apresentados. Nem todos os padrões do GoF [10] foram mostrados, porém, com o conhecimento dos padrões e técnicas nesse livro, acredito que o leitor tenha uma boa base para a realização e evolução do projeto de uma aplicação. Além disso, com essa visão adquirida, fica mais simples a compreensão de outros padrões.

Esse último capítulo não apresenta nenhum padrão novo, porém explora diversos tópicos mais avançados e recentes a respeito da utilização de padrões. Ele aborda, por exemplo, sua utilização para o desenvolvimento de frameworks e como os tipos genéricos podem ser usados de forma efetiva na sua implementação. Também são discutidas questões a respeito da utilização de padrões com a técnica de desenvolvimento Test-Driven Development e como os padrões apresentados se aplicam para

problemas mais amplos relacionados à arquitetura. Finalmente, o livro e o capítulo finalizam falando um pouco sobre a comunidade nacional e internacional de padrões.

10.1 Frameworks

"Um framework não é bom pelo que ele faz, mas quando pode ser utilizado para o que ele não faz."
– Eduardo Guerra

Nos dias de hoje, é muito difícil alguém desenvolver uma aplicação sem a utilização de um framework, principalmente na linguagem Java. Ele não apenas provê um reuso de seu código, como também o reúso de sua estrutura, de seu design. Dessa forma, além de possibilitar o reaproveitamento de funcionalidades acelerando o ritmo de desenvolvimento, ele também direciona a arquitetura da aplicação à utilização de boas práticas de código.

Um framework pode ser visto como um software incompleto que precisa ser preenchido com partes específicas de uma aplicação para poder ser executado [22]. Imagine, por exemplo, um framework que faça o agendamento de execuções. Sozinho, ele não faz nada, pois não tem o que ser agendado sem a existência de uma aplicação. Ao ser instanciado, ele é completado com classes da aplicação que serão invocadas por ele, e então faz sentido a sua execução.

Uma estrutura abstrata referente a um determinado domínio é apresentada pelo framework. Esse domínio pode ser horizontal, referente a uma camada de aplicação, como apresentação ou persistência, ou vertical, referente ao domínio da aplicação, como seguros ou comércio eletrônico. De qualquer forma, o framework captura o conhecimento desse domínio, identificando os principais papeis que as classes desempenham e como elas se relacionam. A partir disso, ele provê um estrutura reutilizável que pode ser usada para a criação de aplicações dentro daquele domínio.

Um framework possui pontos com funcionalidade fixa, chamados de *frozen spots*, e pontos com funcionalidade variável, chamados de *hotspots*. Os *hotspots* são pontos de extensão disponibilizados, onde a aplicação pode inserir parte do seu código. Os *frozen spots* são os que efetivamente provêem a funcionalidade do framework e que coordenam a execução dos *hotspots*. Igualmente importantes, os *hotspots* e os *frozen spots* formam sua arquitetura em conjunto.

Ele propõe uma forma de reúso muito diferente de uma biblioteca de funções ou de classes. Quando uma biblioteca é utilizada, a aplicação é responsável por invocar

a funcionalidade das classes, coordenando sua execução e usando-a como um passo de seu processamento. Quando um framework é utilizado, normalmente é ele quem tem o controle sobre a execução e invoca a funcionalidade das classes da aplicação em pontos específicos. Isso permite que a aplicação reutilize o código e a estrutura do framework, especializando-o para suas necessidades.

Um framework também é diferente de padrões de projeto, pois representa efetivamente uma implementação, um software. Os padrões, como já foi dito durante o livro, já existem como conhecimento, como uma ideia. Porém, muitos padrões são utilizados para viabilizar a criação de *hot spots* na estrutura dos frameworks. Os *hook methods* e as *hook classes*, apresentados respectivamente nos capítulos 2 e 3, são exemplos de técnicas que podem ser utilizadas. Através da herança, a aplicação pode estender uma classe do framework e inserir funcionalidade em um método. Com composição, a aplicação pode implementar uma interface do framework e inserir uma classe para ser invocada em um determinado ponto da execução.

Os tipos de *hot spots* apresentados nessa seção não são os únicos que podem ser utilizados em frameworks. A utilização de reflexão e metadados, especialmente definidos com anotações, tem sido uma prática cada vez mais utilizada em frameworks mais recentes [9]. Porém essas questões já estão além do escopo desse livro.

Ainda existem bibliotecas?

Biblioteca de funções é um conceito que veio da programação estruturada, onde o reúso permitido pelo paradigma de programação era mais restrito. Porém, ainda existem algumas bibliotecas de funções em Java, por exemplo, como as classes `Math` e `Collections`. Em aplicações, essa biblioteca de funções é frequentemente colocada em alguma classe terminada com " `Utils`". Nessa classe, usualmente são incluídos métodos, normalmente estáticos, que são de uso geral da aplicação.

Um conceito de biblioteca inserido na programação orientada a objetos são as bibliotecas de classe. Agora, em vez de termos apenas métodos reutilizáveis, temos classes! Um exemplo de biblioteca de classes muito popular em Java é a Collection API. Diferentemente dos frameworks, as bibliotecas são simplesmente classes reutilizáveis, que não permitem nenhum tipo de extensão. O conceito de biblioteca também é aplicável dentro da estrutura de um framework como um conjunto de classes que compartilha de uma mesma abstração e podem ser inseridas em um de seus hotspots. Por exemplo, em um framework de validação de instâncias em que o algoritmo de validação de uma propriedade é um *hotspot*, faria sentido uma biblioteca de classes com os tipos mais comum de validação, como formato de string,

limites numéricos, entre outros.

Muitas vezes, em aplicações, utilizamos classes como se fossem parte de uma biblioteca, porém se olharmos com calma sua documentação, veremos que ela possui diversos *hotspots* pelos quais é possível estender o seu comportamento, caracterizando-se mais como um framework. Isso tem a ver com um padrão chamado de **Low Surface-to-volume Ratio** [4], cuja tradução do nome seria "baixa proporção da superfície para o volume". Esse padrão tem a ver com você tentar prover uma série de serviços a partir de uma interface externa compacta. Isso torna o framework mais simples de ser utilizado, muitas vezes fazendo parecer que quem está provendo aquele serviço é uma simples classe. O **Facade** é muitas vezes utilizado para se atingir esse objetivo.

Outro fator que faz com que muitas vezes os frameworks se pareçam com simples classes vem do fato dos *hotspots* serem configurados por default com classes mais comuns de serem usadas. Outra alternativa são os *hotspots* serem configurados através de **Builders** que cria e injeta as classes de acordo com parâmetros e configurações. Esse tipo de prática propicia uma **Gentle Learning Curve** [4], ou curva de aprendizado suave, pois o desenvolvedor usuário do framework não precisa conhecer e configurar todos os *hotspots* para instanciar o framework. Porém, caso seja necessário, ele pode ir aos poucos aprendendo mais sobre a sua estrutura para alterar o comportamento quando for necessário.

Outros Exemplos de Frameworks

Se você já desenvolveu alguma aplicação em Java, muito provavelmente já usou algum framework. Muitas vezes os desenvolvedores acabam não percebendo que a classe que estão desenvolvendo faz parte de um contexto mais amplo e é invocada por um framework em um *hotspot*. Essa seção irá apresentar alguns exemplos de framework para que os conceitos apresentados possam ser vistos a partir de uma perspectiva mais concreta.

Vou começar com o exemplo de frameworks para aplicações web, pois além de ser um domínio familiar para muitos leitores, eles possuem um *hotspot* bem óbvio. O que muda de uma requisição para outra? A funcionalidade que é executada quando a requisição chega no servidor! Dessa forma, a classe responsável por tratar uma requisição, chamada de *controller* em um modelo arquitetural MVC [23], acaba sendo um *hotspot* nesse framework. Exemplos de classes desse tipo seriam Servlets na API padrão Java EE, Actions no framework Struts e Managed Beans no JSF. Observe que antes da classe da aplicação receber o controle da requisição, diversas outras funcio-

nalidades são executadas pelo framework, indo desde a tradução dos parâmetros até controle de acesso.

Um outro exemplo interessante de framework é o Quartz (http://quartz-scheduler.org/) , cujo principal objetivo é realizar o agendamento de tarefas. É possível perceber que pelo domínio desse framework, sua execução não faz sentido fora do contexto de uma aplicação. Por mais que ele possua classes implementadas para as mais variadas opções de agendamento, sem a aplicação ele não tem o que agendar. Dessa forma, o principal *hotspot* do framework são as tarefas com lógica da aplicação a serem executadas de acordo com o agendamento realizado. A listagem a seguir mostra um exemplo de uma classe para ser executada.

Listagem 10.1 - Exemplo de classe usada como hotspot do framework Quartz:

```
public class ExcluiUsuariosInvalidos implements Job {
    public void execute(JobExecutionContext context)
      throws JobExecutionException{
          // lógica específica da aplicação
    }
}
```

Um exemplo de framework um pouco diferente é o Log4J, cujo objetivo é fazer o log de uma aplicação. Dentro desse domínio, o Log4J é bastante flexível, permitindo a gravação de diversos tipos de informações, em diferentes formatos e em diferentes locais, como no console, em arquivos, em bancos de dados e até remotamente. Diferentemente dos outros frameworks citados, o Log4J já possui uma biblioteca de classes que podem ser utilizadas em seus *hotspots*. Dessa forma, uma aplicação poderia utilizar o framework sem implementar nenhum de seus *hotspots*, porém se for necessário ela pode, por exemplo, criar classes para adição de propriedades específicas no log ou para que seja realizado o log em um local diferente. A partir das configurações do Log4J, apresentadas na listagem a seguir, é possível perceber que existem várias propriedades que recebem nomes de classe, que na verdade estão configurando implementações para serem usadas nos *hotspots*.

Listagem 10.2 - Exemplo de arquivo de configuração do Log4J:

```
# Configura o nível de log para o logger principal como INFO.
log4j.rootLogger=INFO, Console, LogFile

# Configuração do log do console
```

```
log4j.appender.Console=org.apache.log4j.ConsoleAppender
log4j.appender.Console.layout=org.apache.log4j.PatternLayout
log4j.appender.Console.layout.ConversionPattern=%d{ISO8601}
                                                [%t] %-5p %C - %m%n

# Configuração do log em arquivo
log4j.appender.LogFile=org.apache.log4j.DailyRollingFileAppender
log4j.appender.LogFile.DatePattern='.'yyyy-MM
log4j.appender.LogFile.file=/path/to/logfile.log
log4j.appender.LogFile.layout=org.apache.log4j.PatternLayout
log4j.appender.LogFile.layout.ConversionPattern=%d{ISO8601}
                                                [%t] %-5p %C - %m%
```

Enxergando o GeradorArquivo como um Framework

O exemplo do gerador de arquivo utilizado ao longo desse livro pode ser considerado um framework. Isso porque a classe `GeradorArquivo` não apenas pode ser invocada pela aplicação, como também provê pontos em que essa funcionalidade pode ser estendida e especializada por aplicações. A Figura 10.1 apresenta quais seriam os *hotspots* e os *frozen spots* do gerador de arquivos. Exemplos de *frozen spots* estão no algoritmo de geração de arquivos e na combinação dos pós-processadores no **`Composite`**. Já os *hot spots* seriam o formato de geração dos arquivos, as formas de pós-processamento e a própria ordem de execução dos pós-processadores.

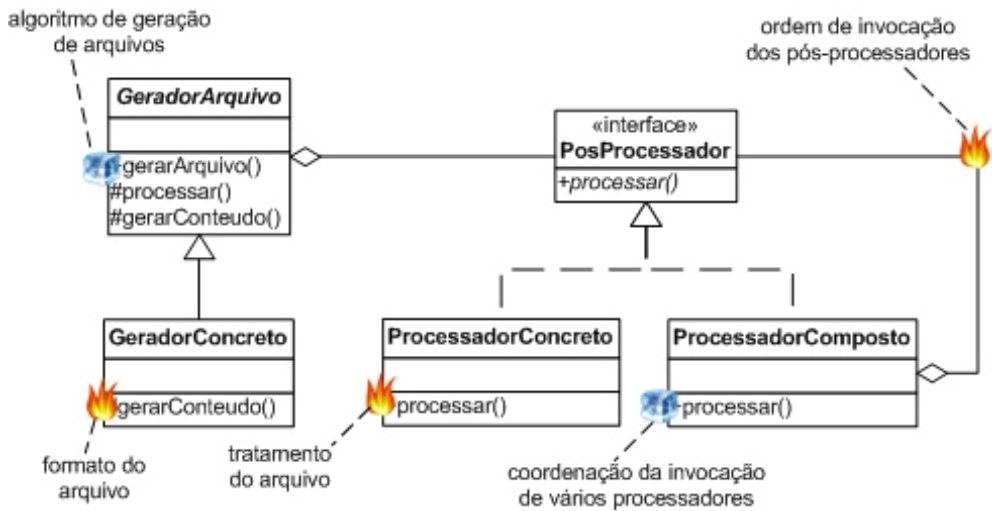



Figura 10.1: Hotspots e frozen spots do gerador de arquivo

Uma coisa importante de se ressaltar é que nem sempre precisa ser uma classe da aplicação a ser utilizada no lugar de um *hot spot*. Um framework pode possuir uma biblioteca de classes que representam implementações mais gerais que podem ser utilizadas pela aplicação em um ponto de extensão. Por exemplo, no gerador de arquivos, o `Compactador` e o `Criptografador` seriam classes mais gerais que poderiam fazer parte da biblioteca de classes do framework. Porém, nada impede que uma aplicação crie um implementação própria e utilize naquele ponto.

10.2 Utilizando Tipos Genéricos com os Padrões

"Eu apenas quero crescer criativamente e ser inspirado. Eu não quero fazer algo genérico ou burro."
– Sienns Miller

Os tipos genéricos são uma funcionalidade de linguagem introduzida no Java 1.5. A partir dela, é possível parametrizar tipos para que seja possível substituir partes da assinatura de seus métodos, como retornos e argumentos, pelo parâmetro passado para o tipo. Um cenário onde isso é muito aplicável é para coleções. Uma lista, por exemplo, ao ser instanciada pode receber como parâmetro o tipo que deve armazenar, de forma que métodos de recuperação retornem aquele tipo e métodos de inserção aceitem como parâmetro apenas aquele tipo.

O suporte de linguagem a tipos genéricos é uma funcionalidade que visa primeiramente à segurança de código. Antes de existirem tipos genéricos, as coleções no Java podiam receber qualquer objeto. O problema acontecia quando uma instância era recuperada e precisava ser convertida para um determinado tipo. Caso o objeto não fosse do tipo esperado, uma `ClassCastException` era lançada. Com tipos genéricos, além do código ficar mais limpo sem a necessidade de fazer o *cast*, erros como esse são pegos já em tempo de compilação.

Os tipos genéricos também podem ser utilizados na definição de interfaces. Dessa forma, a classe que implementar essa interface pode manter o tipo genérico para ser definido nas instâncias ou já definir o tipo utilizado no momento da implementação. A partir disso, uma mesma interface pode ser utilizada para implementação de diversas interfaces na qual apenas o tipo em retornos e parâmetros são alterados. Baseado nesse modelo, outras classes podem utilizar o tipo da interface parametrizado para aceitarem apenas implementações que, por exemplo, retornarem aquele tipo.

Apesar dos padrões serem ideias mais gerais, os tipos genéricos podem ser muito úteis para a criação de implementações mais flexíveis e seguras. Por exemplo, em padrões onde existe a colaboração de duas classes, pode-se garantir em tempo de compilação, a partir do contrato entre as classes, que apenas instâncias com o mesmo tipo genérico podem se relacionar.

Para exemplificar, vamos utilizar o padrão **`Observer`**. Nesse padrão, um observador é aquela classe que possui o papel de receber notificações dos objetos que está observando. Como nessa definição, o objeto passado como parâmetro na notificação pode variar, este acaba sendo um excelente candidato para ser um parâmetro genérico da interface que define esse contrato. A listagem a seguir apresenta a interface `Observador` que define um parâmetro genérico relativo à classe do evento utilizada para notificação.

Listagem 10.3 - Definição de um observador com tipo genérico:

```
public interface Observador<E>{
    public void notificar(E evento);
}
```

Para que o software faça sentido, uma classe deve poder apenas ser observada por uma que consiga entender os eventos gerados por ela. Ou seja, alguém que gera eventos de uma classe só poderia ser observado por alguém que recebe evento dessa

classe, ou de alguma de suas abstrações. A listagem a seguir apresenta a interface `Observavel` que também define um tipo genérico. No caso, ela aceita no método `adicionaObservador()`, uma classe que implemente a interface `Observador` com o tipo genérico igual ou sendo uma abstração ao seu.

Listagem 10.4 - Restringindo os observadores com um tipo genérico:

```
public interface Observavel<E>{
    public void adicionaObservador(Observador<? super E> o);
}
```

A listagem a seguir mostra como as interfaces poderiam ser implementadas e conectadas. Suponha uma aplicação que trabalhe com compras e vendas de ação, na qual existem eventos que representam as operações realizadas. Dentro dessa aplicação, existe um evento mais abstrato chamado `OperacaoAcao`, que possui duas especializações, sendo essas `CompraAcao` e `VendaAcao`.

Listagem 10.5 - Criando e conectando as interfaces com tipos genéricos:

```
public class ObservadorOperacao implements Observador<OperacaoAcao>{
    public void notificar(OperacaoAcao evento){ ... }
}

public class ProcessadorCompra implements Observavel<CompraAcao>{
    public void adicionaObservador(Observador<? super CompraAcao> o){
        ...
    }
}

ObservadorCompra oc = new ObservadorCompra();
ProcessadorCompra pc = new ProcessadorCompra();
pc.adicionaObservador(oc);
```

Nesse contexto, considere uma classe chamada `ObservadorCompra` que implemente a interface `Observer` com o tipo genérico `CompraAcao` e uma classe chamada `ProcessadorCompra` que implemente a interface `Observavel` com o tipo genérico `CompraAcao`. Nesse caso, o observador poderia ser inserido na classe observável, pois uma classe que recebe eventos do tipo `OperacaoAcao` vai saber lidar com um evento do tipo `CompraAcao`, por ser sua superclasse. Porém,

uma classe que, por exemplo, implementasse `Observavel<VendaAcao>` não poderia ser passada para a classe `ProcessadorCompra`, por tratar um tipo de evento diferente do gerado por essa classe.

A utilização de tipos genéricos para implementação de padrões ajuda na criação de soluções que permitem uma maior segurança de código. No exemplo apresentado do padrão **Observer**, a própria estrutura montada impede que uma classe não compatível seja passada como parâmetro. Esse tipo de restrição não seria imposta se o método da interface `Observador` aceitasse um `Object` como parâmetro. Por outro lado, se fosse preciso restringir o tipo observado sem usar tipos genéricos, seria necessária a criação de uma interface para cada caso.

O uso do **Observer** foi apenas um exemplo para mostrar como a utilização de tipos genéricos pode ajudar a amarrar os tipos de uma solução com padrões para evitar inconsistências. Outros padrões também podem ser utilizados para se beneficiar desse recurso de linguagem [1]. O **Abstract Factory**, por exemplo, pode utilizar tipos genéricos para assegurar algum relacionamento entre os tipos retornados e o **Command** pode se utilizar desse recurso para permitir que as classes declarem no tipo o retorno resultante de sua execução.

10.3 Padrões com Test-driven Development

"TDD é uma ideia louca que funciona."
– Kent Beck

O Test-driven Development, TDD, é uma técnica de desenvolvimento e projeto de software na qual os testes são criados antes do código de produção. É uma técnica que vem se tornando cada vez mais popular no mercado e mais atenção da academia. Com essa técnica, as classes vão sendo desenvolvidas incrementalmente em pequenos ciclos, os quais alternam entre a criação de um teste, o desenvolvimento de código de produção para satisfazer o teste adicionado e a refatoração do código criado.

Durante a criação dos testes, o trabalho do desenvolvedor é focado na definição da API externa da classe e do seu comportamento esperado dentro de cada cenário definido. A partir de cada teste adicionado, a tarefa a seguir é implementar a solução mais simples possível na classe que está sendo desenvolvida, de forma ao novo teste e os anteriores serem todos satisfeitos. Como nesse momento o objetivo é fazer o teste passar, logo que isso é atingido, segue um momento para se refletir sobre o design

e a clareza do código. Dessa forma, antes de seguir o desenvolvimento com a introdução de mais um teste, o código é refatorado e a manutenção do comportamento verificada com a execução dos testes.

Existem hoje muitos desenvolvedores que utilizam o TDD mais como uma técnica de desenvolvimento do que uma técnica de projeto. Em outras palavras, eles projetam como serão as classes e como elas irão se relacionar, e depois implementam a funcionalidade utilizando TDD. Quando ele é utilizado também como uma técnica de projeto, a modelagem da aplicação vai acontecendo de forma evolutiva, incluindo a definição das colaborações e dos contratos entre as classes. A seção a seguir fala sobre como o projeto ocorre com TDD e o papel dos padrões nesse processo.

Projetando com TDD e Padrões

O projeto de uma classe com TDD acontece em grande parte na definição do teste. É nele que você vai expressar a API externa da classe e como ela é utilizada pelos seus clientes. O fato dos testes serem criados antes favorece certas características na classe que está sendo desenvolvida, o que favorece a sua testabilidade. Uma dessas características é a coesão, pelo fato de que é mais simples testar uma classe com uma responsabilidade bem definida do que classes que acumulam diversas responsabilidades. Isso ajuda em um melhor particionamento das classes para uma melhor divisão de responsabilidades.

Outra questão que é trabalhada durante os testes é o desacoplamento da classe que está sendo desenvolvida das suas dependências. Apesar de não ser obrigatório no TDD, normalmente utilizam-se testes de unidade no desenvolvimento. Isso é feito para que o teste possa focar exclusivamente nas responsabilidades daquela classe. Para isso, é necessário inserir objetos falsos (mock objects) na classe testada, e simular diferentes comportamentos das dependências para a criação dos cenários de teste.

Para possibilitar a inserção dos mock objects, é preciso que a classe possua um mecanismo no qual as dependências possam ser substituídas. Além disso, é necessário que seja definido um contrato, normalmente através de uma interface, que formalize a interação entre a classe desenvolvida e as dependências. Essas necessidades de teste resultam na utilização de um conjunto de boas práticas que acabam favorecendo o design da aplicação.

Mas será que apenas esses princípios de TDD são suficientes para modelar um software? Na verdade, os testes servem como um mecanismo para expressar o design

desejado para uma classe, mas se o desenvolvedor não souber qual deve ser a solução para o problema, a utilização dessa técnica ficará restrita. O mesmo se aplica para a modelagem com o uso de UML, pois ela é apenas uma ferramenta para expressar o modelo da aplicação através de diagrama, e de nada adianta se o desenvolvedor não souber quais classes vai representar. Sendo assim, não importa qual a forma ou técnica na qual o design da aplicação está sendo definido, os padrões são importantes para que o desenvolvedor possa direcionar sua solução.

No caso do TDD, já no primeiro teste, o desenvolvedor precisa definir como a classe será criada. Nesse momento, já pode entrar em ação o conhecimento a respeito de padrões de criação. Será utilizado um construtor ou um **Static Factory Method**? É preciso utilizar um **Singleton** para a aplicação ter apenas uma instância dessa classe? Em um segundo momento, caso o processo de criação das classes for complexo o suficiente, pode-se partir para a criação de um **Builder**, o qual poderia ser desenvolvido em sua própria classe de teste.

O **Dependency Injection** é um padrão muito utilizado quando se usa TDD, principalmente quando realmente procura-se isolar a classe testada de suas dependências. Isso é feito para facilitar a substituição da dependência por mock objects pelo teste. Porém, como foi visto no Capítulo 7, o padrão **Service Locator** poderia também ser utilizado nessas situações sem prejuízo à modularidade.

Quando um padrão é utilizado, normalmente existem mais de uma classe envolvida, sendo cada uma com uma responsabilidade bem definida. Ao se optar por desenvolver uma solução baseada em um padrão a partir de TDD, deve-se a partir dos testes direcionar a interface da classe e de suas dependências para o uso desse padrão. Imagine um exemplo em que o padrão **Visitor** estivesse sendo utilizado. Nos testes de classes que implementam a interface visitante, que é passada como parâmetro, seria verificado se as chamadas aos métodos gerariam o efeito esperado. Nos testes das classes visitadas, os testes verificariam se os métodos do objeto visitante passado como parametro seriam invocados corretamente. Esse é um exemplo no qual o padrão escolhido interfere na forma como os testes seriam criados. Mesmo assim, a interface que serve como contrato entre essas duas classes poderia ir sendo modelada de forma incremental através do TDD.

A refatoração também é uma fase do ciclo de TDD que é muito importante para o projeto do software que está sendo desenvolvido. Como no TDD o projeto ocorre de forma evolutiva e de acordo com as necessidades correntes, a reestruturação da solução que está sendo adotada é algo feito com uma certa frequência. Nesse contexto, como nem sempre a necessidade do uso de um padrão acontece na primeira

versão da classe, é natural que eles sejam implementados a partir de refatorações. Principalmente quando uma classe começa a acumular responsabilidades ou aparece uma duplicação de código. Ter os padrões como um direcionamento para as refatorações, como foi mostrado no decorrer dos capítulos, é uma excelente forma de dirigir o design da aplicação para boas soluções de forma evolutiva.

Concluindo, por mais que o TDD favoreça a criação de classes mais coesas e desacopladas, isso não é o suficiente para o projeto de uma aplicação como um todo. Nesse contexto, a utilização de padrões, tanto para direcionar os testes, quanto para o alvo de refatorações, pode ter um impacto positivo nas soluções desenvolvidas.

Exemplo de TDD usando o Padrão Observer

Para ilustrar a utilização de padrões com TDD, vamos imaginar o exemplo de uma aplicação de e-commerce. Nela, ao se adicionar um produto no carrinho de compras, existem diversas funcionalidades que devem ser acionadas. Por exemplo, deve-se registrar essa informação para as estatísticas do produto, deve-se adicionar a categoria do produto nos interesses do cliente e deve-se criar uma reserva para uma unidade do produto no estoque. A inclusão de todas essas funcionalidades no carrinho de compras poderia sobrecarregar essa classe e torná-la acoplada a diversos subsistemas.

Um padrão adequado para esse tipo de problema é o **`Observer`**, pois o carrinho de compras poderia notificar as classes interessadas em seus eventos sem estar acoplado a elas. Nesse caso, a responsabilidade da classe que representa o carrinho de compras seria simplesmente notificar os observadores dos eventos ocorridos. Baseando-se nisso, os testes dessa classe deveriam apenas focar se as notificações estão sendo feitas na hora e com as informações corretas.

O primeiro passo seria a criação de um teste que definisse como seria a funcionalidade de notificação do carrinho de compras, conforme o código mostrado na próxima listagem. Observe que é instanciada uma classe, chamada de `MockObservador` para simular a existência de um observador durante o teste. Esse mock object é adicionado como um observador na classe testada e o teste verifica se, ao adicionar um produto no carrinho de compras, o observador é notificado com o produto adicionado.

Listagem 10.6 - Teste que define padrão Observer e verifica recebimento de notificação:

```
@Test
```

```
public void testeNotificacao(){
    MockObservador mock = new MockObservador();
    CarrinhoCompras cc = new CarrinhoCompras();
    cc.addObservador(mock);

    Produto p = new Produto("Cabo HDMI", 30.0);
    cc.adicionar(p);

    assertTrue(mock.recebeuNotificacao());
    assertEquals(p, mock.produtoRecebido());
}
```

Em seguida, é preciso definir a classe `MockObservador` que está sendo utilizada no teste. Essa classe deve implementar a interface esperada pelo método `addObservador()` da classe testada, no caso `ObservadorCarrinho`. Observe que essa classe também deve possuir os métodos utilizados no teste para verificação, no caso `recebeuNotificacao()` e `produtoRecebido()`. Veja que a implementação dessa classe deve fazer sentido para o teste, ou seja, o método `notificaProduto()` exigido pela interface implementada deve armazenar o resultado para poder ser verificado no teste.

Listagem 10.7 - Mock object que simula um observador:

```
public class MockObservador implements ObservadorCarrinho {

    private Produto p;

    @Override
    public void notificaProduto(Produto p){
        this.p = p;
    }
    public boolean recebeuNotificacao(){
        return p != null;
    }
    public Produto produtoRecebido(){
        return p;
    }
}
```

O que é interessante notar nesse caso, é que a interface e os seus métodos são definidos no momento em que está sendo criado o teste. O teste é um mecanismo

que está sendo utilizado para a definição desse contrato, porém é a partir do padrão **Observer** que a solução foi construída.

Nesse caso, após a definição dos testes da classe observada, seriam criadas as classes observadoras, que implementariam a interface `ObservadorCarrinho`. Se o TDD continuasse a ser utilizado nesse desenvolvimento, seria criada uma nova bateria de testes para cada classe. Nesses testes, o método `notificaProduto()` seria invocado simulando uma notificação de uma classe que está sendo observada e verificando o comportamento esperado da classe em questão. Quando as classes estivessem desenvolvidas, um teste de integração, envolvendo o carrinho de compras e as classes observadoras, também poderia ser criado para verificar se as funcionalidades se integram de forma correta.

10.4 Posso Aplicar esses Padrões na Arquitetura?

"Não existem regras na arquitetura para um castelo nas nuvens."
– G. K. Chesterton

Muitos dos padrões apresentados nesse livro também podem se aplicar em um contexto mais amplo de uma arquitetura. Nesse caso, em vez de classes, os participantes dos padrões seriam componentes e até mesmo subsistemas de um software. Como os padrões são ideias e não estão ligados a implementações específicas, os problemas e as soluções dos padrões podem ser aplicados em uma escala diferente.

Pegando novamente como exemplo o padrão **Observer**, da mesma forma que uma classe pode precisar receber notificações de outra, um subsistema pode precisar receber atualizações de outros subsistemas remotos. Enquanto dentro de um software orientado a objetos os contratos são firmados a partir de interfaces e classes abstratas, entre sistemas remotos são utilizados protocolos de comunicação. A solução certamente precisa de uma adaptação, mas certamente a mesma ideia pode ser usada.

Porém, a utilização desses padrões deve ser feita com muito cuidado, pois apesar das soluções serem similares, as consequências de seu uso podem ser bem diferentes. Por exemplo, em uma arquitetura, quando lidamos com subsistemas que estão distribuídos em uma rede, sempre devemos considerar a possibilidade de um dos nós envolvidos falhar. Isso é algo que não precisamos considerar quando estamos lidando com classes implantadas em uma mesma máquina virtual. Essa é apenas uma das questões que devem ser consideradas; existem outras, como a tolerância a esse

tipo de falha, questões de desempenho em relação à forma como é feita a comunicação, entre outras.

Para exemplificar essa questão, considere o padrão **Proxy**. Como foi visto, ele pode ser utilizado para proteger o acesso a um determinado objeto. Normalmente, quando isso é feito, é adicionada uma funcionalidade nesse acesso, como, por exemplo, alguma verificação de segurança ou validação dos dados. Dessa forma, o **Proxy** fica entre o cliente e a classe que está sendo acessada.

De forma similar, o **Proxy** também pode ser aplicado de forma mais abstrata a arquiteturas. Nessa caso, ele irá ficar como intermediário entre dois subsistemas. Ele irá disponibilizar os mesmos serviços e utilizar o mesmo protocolo que o subsistema original, para que seja transparente para o cliente a sua existência. Em outras palavras, o cliente deve acessar o **Proxy** como se estivesse acessando o subsistema original. A Figura 10.2 mostra a representação desse padrão implementado no contexto de uma arquitetura. Nesse caso, **Proxy** também poderia adicionar alguma funcionalidade nesse acesso, como de segurança ou registro de auditoria, da mesma forma que o padrão original.

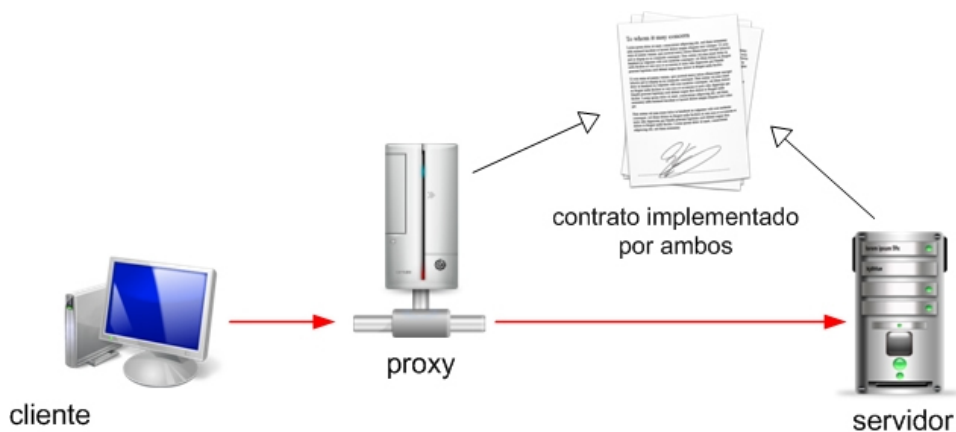



Figura 10.2: Estrutura de um Proxy para Criação de Arquitetura

Porém, como foi dito, o uso do **Proxy** em uma arquitetura pode ter consequências diferentes de seu uso com objetos. Por exemplo, se houver algum problema com a máquina onde estiver o **Proxy**, o cliente perderá acesso ao serviço, mesmo este estando em funcionamento. Isso adiciona um novo possível ponto de falha na arquitetura, o que é uma consequência negativa em relação à implementação com objetos.

Por outro lado, o `Proxy` também pode ser responsável pelo roteamento das mensagens entre o cliente e o servidor, permitindo que o servidor mude sua localização sem afetar os clientes, o que seria uma nova consequência positiva.

Muitas vezes, é complicado separar o que é uma solução de design do que é uma solução arquitetural. O importante é não utilizar as soluções dos padrões cegamente e sempre considerar as consequências antes da sua utilização. Lembre-se que existem soluções alternativas que podem resolver o problema de uma forma diferente, gerando um outro conjunto de consequências. Trabalhar com arquitetura é saber fazer trocas, por exemplo, sacrificando desempenho para aumentar a segurança, e escolher as soluções de forma a atender os requisitos não-funcionais do sistema.

10.5 Comunidade de Padrões

"Fundamental a qualquer disciplina relacionada a ciência ou engenharia é uma vocabulário para expressar seus conceitos, e uma linguagem para os relacionar juntos."
– Brad Appleton

Após ler esse livro, você pode se perguntar: Como surgem novos padrões? Onde eu encontro novos padrões? Como eu faço para trabalhar com isso? O Hillside Group é uma organização internacional sem fins lucrativos cujo objetivo é fomentar a documentação de padrões para as mais diferentes áreas, principalmente as relacionadas ao desenvolvimento de software. Dessa forma, ela auxilia na organização de diversas conferências focadas em padrões ao redor do mundo e ajuda autores a amadurecerem seus padrões para sua publicação, seja em formato de artigos ou de livros.

As conferências de padrões, conhecidas como PLoP (Pattern Languages of Program), acontecem ao redor do mundo e recebem submissões de artigos com padrões, linguagens de padrões, ou relativos a utilização de padrões. A conferência principal acontece todo ano nos EUA e em 2013 irá completar 20 edições. Pessoalmente já tive o prazer de participar de algumas edições, e inclusive de ser o organizador do PLoP 2012, que aconteceu em Tucson, no Arizona. São conferências normalmente para um número pequeno de pessoas (normalmente entre 40 e 60 pessoas) e que são muito diferentes de uma conferência científica tradicional.

Existem outras conferências PLoP ao redor do mundo como o EuroPLoP (Alemanha), AsianPLoP (Japão), KoalaPLoP (Austrália), VikingPLoP (países da Escandinávia), e, o estreante de 2013, GuruPLoP (Índia). Existem também conferências

temáticas, onde pessoas interessadas se reúnem para discutir padrões de um domínio específico, como o Meta PLoP (metaprogramação), o Scrum PLoP (padrões para documentar práticas do Scrum) e o ParaPLoP (programação paralela). No Brasil, a cada dois anos, ocorre o SugarLoafPLoP, o evento latino-americano de padrões, e nos anos intermediários um MiniPLoP, uma versão reduzida do evento.

O que são linguagens de padrões?

Uma linguagem de padrões é um conjunto de padrões que documentam soluções e boas práticas para um determinado domínio. Por exemplo, eu escrevi uma linguagem de padrões para a construção de frameworks que utilizam anotações e metadados. Mais do que simplesmente uma solução pontual, a linguagem de padrões possui soluções que sinergicamente podem ser combinadas e se complementam, sendo uma forma de documentar o conhecimento de modelagem sobre aquele domínio.

A submissão de um padrão para um PLoP possui um processo diferente do que a submissão de artigos para outras conferências. Ao submeter um padrão, após uma filtragem inicial, ele é atribuído a um *shepherd* (pastor). O *shepherd* é uma especialista na área de padrões cujo papel é orientar e ajudar o autor a evoluir e amadurecer o trabalho submetido. Sendo assim, durante cerca de um mês e meio, o autor em conjunto com o *shepherd* irão interagir e cooperar para que o trabalho evolua durante esse período. Só depois disso que o artigo realmente será avaliado pelo comitê e decidido se ele será aceito ou não na conferência.

Na conferência, o autor precisa participar de uma seção chamada de *Writers Workshop*, que é bem diferente de uma seção tradicional onde o autor simplesmente apresenta o seu trabalho. Nessa seção, um grupo irá discutir o trabalho do autor levando em consideração o entendimento que obtiveram sobre o trabalho, apontando pontos de melhoria e dando sugestões. O autor participa da discussão apenas no inicio, quando se lê uma pequena parte do artigo, e no fim, quando pode tirar dúvidas sobre algum comentário. Por mais angustiante que seja ver as pessoas discutindo sobre seu trabalho e não poder opinar ou explicar alguma coisa, esse é um processo com o qual se tem um grande feedback a respeito de como outras pessoas enxergam o seu trabalho.

Pessoalmente gosto muito do estilo da comunidade de padrões, que tem um es-

pírito muito colaborativo, focado em ajudar aqueles que participam dela a documentarem seus padrões da melhor forma possível. Nas conferências, além do estilo peculiar das apresentações, ainda existe toda uma cultura que valoriza muito a interação entre as pessoas. Existem, por exemplo, tradições como a de jogos para aproximar os participantes e troca de lembranças em que cada um traz normalmente algo típico de sua região ou país.

10.6 E agora?

"Toda vez que você tem medo de fazer alguma coisa e você faz, ela te deixa mais forte. Mesmo que você falhe."
– Fred Bartlit

Durante esse livro foi trilhado um caminho que passou por diversos padrões que mostram e exemplificam técnicas para a criação do projeto de um software orientado a objetos. Diferentemente de quando se aprende a usar um novo framework ou uma nova API, quando se está adquirindo conhecimento, saber projetar um software é uma habilidade. E uma habilidade só se aprende de verdade com prática, sendo que o conhecimento é necessário, porém não suficiente.

Sendo assim, o próximo passo agora é tentar colocar em prática os padrões e técnicas apresentados neste livro. Quando desenvolver um software procure olhar para ele com olhar mais crítico, tentando identificar quais padrões ele já implementa, muitas vezes de forma induzida pelos frameworks utilizados. Uma dica pode estar nos próprios nomes das classes, que com frequência "entregam" o padrão utilizado. Procure raciocinar em cima daquela solução e identificar quais foram os motivos para a utilização daquele padrão e quais benefícios ele está trazendo para o design. Aprendemos muito observando soluções prontas e abstraindo o que foi utilizado na sua concepção.

Em seguida, veja quais problemas você consegue identificar e como poderia ser feito para melhorar. Muitas vezes, um padrão deixou de ser aplicado em uma situação em que ele era adequado, porém também não é raro encontrar implementações de padrões incorretas, em que eles foram utilizados no contexto errado. A duplicação de código, por exemplo, é um indício de que algo não está cheirando bem naquela parte do software. Veja então como algum padrão poderia te ajudar na resolução daquele problema, considerando as consequências e qual o peso de cada uma para o cenário em questão.

Adicionalmente, ao criar uma nova solução, pense em como algum padrão poderia te ajudar na resolução dos problemas envolvidos. Procure conter a ansiedade de aplicar diversos padrões em sequência sem ter alguma motivação nos requisitos. Também não deixe de combinar os padrões nas situações em que isso for adequado. Projetar um software consiste em saber equilibrar os requisitos através da combinação de soluções para os pequenos problemas, de forma a se obter um resultado final adequado. Teste suas soluções! Crie cenários baseados em histórias do cliente e veja se seu design se encaixa. Use boas práticas e mantenha o código limpo para que seja fácil de refatorá-lo frente a novas necessidades.

Finalmente, desenvolvimento de software é algo que se estuda durante toda sua carreira! Conheça novos padrões, estude como os padrões se encaixam nas novas tecnologias, e busque os padrões específicos para os domínios relacionados a sua arquitetura. Além disso, lembre-se que design de software é uma atividade criativa, então evite formulas prontas, tenha sempre uma visão crítica e utilize os padrões como uma ferramenta da sua criatividade.

Boa sorte!

Índice Remissivo

Referências Bibliográficas

[1] Alex Marques Campos Alexandre Gazola. Padrões de projeto com generics - revista mundoj, edição 35.

[2] Kent Beck. *Implementation Patterns*. Addison-Wesley Professional, 2007.

[3] Joshua Bloch. *Effective Java, Segunda edição*. Addison-Wesley, 2008.

[4] Joseph W. Yoder Brian Foote. The selfish class - pattern languages of program design 3. 1997.

[5] Murray Silverstein Christopher Alexander, Sara Ishikawa. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[6] John Crupi Deepak Alur, Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2003.

[7] Scott Stanlick Don Brown, Chad Michael Davis. *Struts 2 in Action*. 2008.

[8] Ralph Johnson Don Roberts. Evolving frameworks: A pattern language for developing object-oriented frameworks - proceedings of the third conference on pattern languages and programming. 1996.

[9] Uirá Kulesza Clovis Fernandes Eduardo Guerra, Felipe Alves. A reference architecture for improving the internal structure of metadata-based frameworks - journal of systems and software, volume 86, issue 5. 2013.

[10] Ralph Johnson John Vlissides Erich Gamma, Richard Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[11] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.

[12] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[13] Martin Fowler. Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html, 2004.

[14] Martin Fowler. Fluent interface. http://martinfowler.com/bliki/FluentInterface.html, 2005.

[15] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[16] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley Professional, 2004.

[17] Joseph W. Yoder Leon Welicki, Rebecca Wirfs-Brock. The dynamic factory pattern - proceedings of the 15th conference on pattern languages and programming. 2008.

[18] James Noble. Classifying relationships between object-oriented design patterns - australian software engineering conference. 1998.

[19] Clifton Nock. *Data Access Patterns: Database Interactions in Object-Oriented Applications*. Addison-Wesley Professional, 2003.

[20] Larry O'Brien. Design patterns 15 years later: An interview with erich gamma, richard helm, and ralph johnson. http://www.informit.com/articles/article.aspx?p=1404056, 2009.

[21] Java Community Process. Jsr-315 javatm servlet 3.0. http://jcp.org/aboutJava/communityprocess/final/jsr315/index.html, 2009.

[22] Brian Foote Ralph Johnson. Designing reusable classes - journal of object-oriented programming. 1988.

[23] Trygve Reenskaug. The model-view-controller (mvc) - its past and present. http://heim.ifi.uio.no/~trygver/2003/javazone-jaoo/MVC_pattern.pdf, 2003.

[24] Nat Pryce Steve Freeman. Evolving an embedded domain-specific language in java - proc. oopsla 2006. 2006.

[25] Tim Mackinnon Joe Walnes Steve Freeman, Nat Pryce. Mock roles, not objects - proc. oopsla 2004. 2004.

[26] Ovidiu Tudosa. Prevalent system. https://wiki.engr.illinois.edu/download/attachments/121733136/prevsystem_otudosa2.pdf, 2009.

[27] Bobby Woolf. The null object pattern - pattern languages of program design 3. 1997.

[28] Michel de Champlain Yun Mai. A pattern language to visitors - proceedings of the 8th conference on pattern languages and programming. 2001.